Problem 1. In this problem we will work through the task of computing the point of reflection of two points with respect to a line in 2-dimensional space. Let $a = (a_x, a_y)$ and $b = (b_x, b_y)$ be two points in the plane. Let $\ell$ denote the line passing through these points. We are also given two points $p = (p_x, p_y)$ and $q = (q_x, q_y)$ both of which lie on the same side of $\ell$ (see Fig. 1(a)).

(a) Let $\vec{v} = b - a$. We know that any point on line $\ell$ can be expressed as an affine combination $\alpha_1 a + \alpha_2 b$ for reals $\alpha_1$ and $\alpha_2$, where $\alpha_1 + \alpha_2 = 1$. We can also express any point on this line (in ray-shooting form) as $a + t \vec{v}$ (for any $t \in \mathbb{R}$). What is the relationship between $(\alpha_1, \alpha_2)$ and $t$?

(b) Let $\vec{u} = q - a$ be the vector directed from $a$ to $q$. Explain how to compute the orthogonal projection $\vec{u}'$ of $\vec{u}$ onto $\vec{v}$ and its orthogonal complement $\vec{u}''$ (see Fig. 1(b)).

(c) Using these vectors, show how to compute the point $q'$ which is the reflection of $q$ about $\ell$. In particular, derive the coordinates $q' = (q'_x, q'_y)$ (see Fig. 1(c)).

(d) By basic geometry, the desired point $r$ is the intersection of the line segment $pq'$ with $\ell$. Based on the quantities computed thus far, explain how to compute $r$’s coordinates. (Hint: We have already seen that $r = a + t(b - a)$ for some real $t$ and analogously we have $r = p + s(q' - p)$ for some real $s$. Solve these two simultaneous equations to determine the value of $t$ as a function of the coordinates of $a, b, p$ and $q'$.)

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

- **drawFoot()**: Draws a foot with the ankle centered at the origin extending along the $x$-axis.
drawCalf() : Draws a calf with the ankle centered at the origin extending up along the $y$-axis. The knee is located 12 units above the origin.

drawThigh() : Draws a thigh with the knee centered at the origin extending up along the $y$-axis. The hip is located 10 units above the origin.

Using these two procedures, implement in OpenGL a drawing procedure

drawLeg($x$, $y$, $\theta_1$, $\theta_2$, $\theta_3$)

which draws the leg so that the hip is placed at the coordinates ($x$, $y$), the thigh is rotated about the hip by the angle $\theta_1$ (positive angle corresponds to a counterclockwise rotation), the calf is rotated about the knee by the angle $\theta_2$ (positive angle corresponds to a clockwise rotation), and the foot is rotated about the ankle by angle $\theta_3$ (positive angle corresponds to a counterclockwise rotation). (See Fig. 2(b).) Note that angles are relative, not absolute. That is, changing $\theta_1$ affects the rotation of all the limbs, changing $\theta_2$ affects the rotations of both the calf and foot, etc. All angles are given in degrees.

On return from your procedure, the Modelview matrix stack should be unchanged.

Problem 3. In class we showed how to compute the specular component of the Phong illumination model. This assumed that the light source was a point. If instead the light source has area, then the specular region will be some expansion of the reflection of the light source on the surface. In this problem, we will consider how to compute this.

Figure 2: Drawing a leg.

Figure 3: The reflection of a triangular light source.
Suppose that you have a coordinate system in 3-dimensional space in which the $x, y$-coordinate plane (that is, the plane $z = 0$) is horizontal and the $z$-axis points up vertically. Define $x, y$-coordinate plane to be the ground surface. You are given a triangular light source $\triangle p_1 p_2 p_3$ and a viewer located at a point $q$ (see Fig. 3(a)). You may assume that both the light source and viewer lie entirely above the ground. The reflection of the light source will appear to the viewer to be a triangular region on the ground.

(a) Given two points $p = (p_x, p_y, p_z)$ and $q = (q_x, q_y, q_z)$, both lying above the ground, present (as a function of the coordinates of $p$ and $q$) the coordinates of a point $r = (r_x, r_y, r_z)$ on the ground at which $q$ sees the reflection of $p$ (see Fig. 3(b)). (This is similar to Problem 1, but there is a simpler solution because of the assumption that the reflection surface is on the plane $z = 0$.) Briefly explain how you derived your answer.

(b) Using the result from part (a), show that given $q$, there exists a projective transformation $T$ such that $r = T(p)$. More formally, show that there exists a $4 \times 4$ matrix $T$ such that, assuming that $r$ and $p$ are expressed in projective homogeneous coordinates, after perspective normalization, $T \cdot p = r$. That is,

$$T \cdot p = \begin{bmatrix} t_{1,1} & t_{1,2} & t_{1,3} & t_{1,4} \\ t_{2,1} & t_{2,2} & t_{2,3} & t_{2,4} \\ t_{3,1} & t_{3,2} & t_{3,3} & t_{3,4} \\ t_{4,1} & t_{4,2} & t_{4,3} & t_{4,4} \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} r'_x/t'_w \\ r'_y/t'_w \\ r'_z/t'_w \\ 1 \end{bmatrix} \equiv \begin{bmatrix} r_x \\ r_y \\ r_z \\ 1 \end{bmatrix}$$

(Note that the entries of $T$ can depend on $q$, but not on either $p$ or $r$.) Briefly explain how you derived $T$.

Although I will not ask you to finish this, you have the basic ingredients for computing the specular point. In particular, you can load this matrix on the GL_PROJECTION stack, and then draw the light source. The matrix will project the vertices of the light source onto the ground surface.

**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.

Many computer models involve objects that are rotationally symmetric with respect to some axis. As an example, see the model of a chess piece shown in Fig. 4(a). The vertices of such a model can be presented as a sequence of points $(p_0, \ldots, p_n)$, each of which consists of an $x$ and $z$ coordinate, that is $p_i = (x_i, z_i)$ (see Fig. 4(b)).

This defines a sequence of $n$ slices, where slice $i$ involves a circular strip of quads rotating about the $z$-axis an passing through points $p_{i-1}$ and $p_i$ on the $x, y$-coordinate plane (see Fig. 4(c) and imagine the $y$-axis is pointing out from the paper). The number $m$ of quads in each circular strip is called the number of slices. (In Fig. 4(c) there are 16 slices.)

Present pseudo-code for a function which, an array of 2-dimensional points $(p_0, \ldots, p_n)$, the value $n$ (which determines the number of slices), and number of slices $m$, outputs the following:

- An OpenGL triangle fan connecting $p_0$ to the vertices of level $p_1$.
- A sequence of $n - 2$ OpenGL quad strips connecting level $p_{i-1}$ level $p_i$, for $2 \leq i \leq n - 1$.
- An OpenGL triangle fan connecting $p_n$ to the vertices of level $p_{n-1}$.
Notes: You may assume that $p_0$ and $p_n$ both lie on the $z$-axis. It is sufficient to output the vertices of the model ($\text{glVertex}$). I do not need the surface normals ($\text{glNormal}$). In order that front sides of the faces of the model be on the outside, the first two vertices of each triangle fan or quad strip should be generated in counterclockwise order about the associated triangle or quad.
Programming Assignment 1: The Most Amazing Ultimate Bubble Shooter

Handed out: Thu, Sep 19. Due: Wed, Oct 2, 11:59:59pm. Late policy: up to 6 hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for each additional 24 hours.

Overview: The goal of this assignment is to learn the basics of OpenGL and GLUT and two-dimensional geometry. 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 essential elements outlined below.

There is a well-known genre of 2-dimensional games, called bubble shooters (see Fig. 1). Your boss has determined that the universe does not yet have enough variants of this style of game, and has asked you to write one. There is a collection colored disks (or bubbles) at the top of the window and a gun that shoots a new bubble of a specified color. The bubble shot from the gun attaches itself to wherever it hits the collection. If the new bubble forms a connected group of three or more bubbles, they are all destroyed. Further, any group of bubbles that becomes disconnected from the group as a result of this are destroyed as well. After a given number of shots, the whole assembly of bubbles moves down one level, and the top row is filled with a new set of randomly colored bubbles. The player’s job is to remove bubbles before any of them hit the bottom of the window. (This is an incomplete description, and you have some flexibility in how you define the game semantics, assuming you implement the basic features described below.)

Figure 1: A screen-shot of a popular bubble shooter game.

Your boss has placed two additional constraints on you. Corporate has informed us that our users are tired of the same window size, and so your game window should be capable of being resized. When the window is resized, the bubbles do not change in size or shape, rather the game is restarted but with a new set of bubbles that fills up the available space. (Thus, a tall window will have more rows than columns, and a wide window will have more columns than rows.) Second, your boss wants you to animate the destruction of bubbles in a different way than normal. In most bubble games, the bubbles simply explode (or pop). Corporate informs us that our users are tired of wimpy bubbles, and they

---

1I have omitted the facts that you can bounce bubbles off the side walls and that the assembly moves down after a given number of shots that result in new bubbles being added. Shots that destroy bubbles do not count against you.
want more robust bubbles. In your game, the destroyed bubbles will appear to be knocked off the background, and they will then fall (as if pulled by gravity) down through the bottom of the window. (We will show an example of this in our demo program.)

The Bubbles: The collection of bubbles, called the board, are arrayed in rows at the top of the window. (I filled half the window in my implementation.) Each bubble can be rendered as a colored circular disk. The array can be a square grid (and you can implement the traditional hexagonal grid for extra credit points). I believe that the traditional implementation used five colors. I used six colors in mine. (If you really want to be fancy, you could make the number of colors depend on the number of columns, since as the window gets narrower, it is hard to make groups of three.)

I implemented my board as a 2-dimensional array of integers (actually an enumeration), where each integer indicates a color, and I used a special value was used to indicate an empty square. You may use whatever representation you like. As mentioned above, the entire array of bubbles moves down periodically. You may define what this means (based on number of shots, number of unproductive shots, or based on elapsed time).

Cannon: The cannon sits near the center of the bottom edge of the window. (The exact location is up to you.) It pivots based on the location of the mouse. (You can use the GLUT passive mouse motion callback to track its location.) When the left mouse button is clicked, it shoots a bubble of a random color. This bubble travels along a straight line, which you should animate. It stops in the last grid position before hitting the bubbles on the board. (For the basic project, bubbles can disappear if they hit the side of the window before hitting any bubble. For extra credit, you can have them bounce off the sides.)

You should indicate to the player what the color of the upcoming bubble will be. For example, this can be done by drawing the future ball in the lower left corner of the window.

Destroyed bubbles: When the shot bubble finally stops, you can perform a simple depth-first or breadth-first search to determine which other bubbles of the same color are connected to it. (In my implementation, I took “connected” to mean one of the four neighbors: north, south, east, and west.) To avoid looping infinitely, I marked each one of these as a “candidate” for destruction. If the number of candidate bubbles is three or more, they are all destroyed.

If the candidate bubbles are to be destroyed, you will also need to determine which bubbles become disconnected from the group. I did this also through a simple depth-first search. All the bubbles on the topmost row were marked as “safe.” Then the search recursively marks all their neighbors as alive, provided they have not already been marked as candidates for destruction. In the end, the bubbles that have been disconnected from the group will not be visited. Any bubble not marked as safe is then destroyed. Of course, you may choose to implement this in any other way you like.

Animating the Destruction: The destroyed bubbles do not simply disappear. I had each one fly up in the air in a random direction (as if exploded off the board), and then they fall as if pulled by gravity. To implement this, you can associate each of these flying bubbles with a state, consisting

\[ \text{OpenGL/GLUT does not have a built-in procedure (that I know of) for drawing circular disks. I wrote a program for drawing an} \ n \text{-sided regular polygon inside a unit circle, by generating} \ n \text{angles from 0 to } 2\pi, \text{and placing each vertex at } (\cos \theta, \sin \theta). \text{Setting } n = 12 \text{ already looks pretty smooth. I then used} \ \text{glScale and glTranslate to scale and position the disk. (For extra credit, find out how to load an image of a circular disk.)} \]
of its current position and its current velocity (a vector quantity). With each time step, move the bubble by its current velocity, and then to simulate gravity, decrement the y-coordinate of the velocity vector by some fixed amount (or generally, some quantity that depends on the amount of elapsed time). This will cause the speed of descent to increase over time, which is just what you want. Once they leave the window, they can be removed for good.

**Smooth animation:** I would recommend using `glutIdleFunc()` to continuously update the state of your game. You can also use the timer callback. If you use the idle function, keep in mind that the game may play at different rates on your machine versus the TA’s machine. Consequently, it is a good idea to use a system function like `ftime` to track the actual elapsed time, and update your object motions accordingly. (See the class web page (under the Projects tab) for further information about `ftime`.)

**Resetting and Quitting:** It should be possible to reset the game to its starting point, say by hitting the ‘r’ key and to quit the game, say by hitting the ‘q’ or ESC keys.

**Resizing the Window:** Your program should allow the user to resize the game window. When the window is resized, a new set of bubbles should be generated to fill up the current window. The bubble sizes and their spacing should remain constant, the number of rows and columns will change. (It is probably easiest to left-justify them, but for extra credit try to center them within the window.) You may assume that there is some reasonable minimum size (e.g., you can assume there will be enough space for at least a couple of rows and a couple of columns.)

**Losing:** The game is lost when some bubble from the board hits the bottom row. Rather than simply exiting, your program should display a special window (possibly with an informative message) and allow the user to restart the game.

**Final Submission:** Submissions will be made through the submit server, [https://submit.cs.umd.edu/](https://submit.cs.umd.edu/). (The submit server is used only for uploading. All testing will be done by the TA.) 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. If you are using MacOS with XCode (which the TA does not have), be sure that you provide directions for compiling and running your program from a regular Unix-like command window. In particular, providing a Makefile (which has variables for the various include and library directories) will be a big help to the TA.

**Makefile or Solution files:** Include any files or instructions needed for compiling your program. (E.g. a Makefile if you are on a Unix system or the `.sln` and `.vcproj` (or `vcxproj`) 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).

Omit: Omit (especially large) binary files that are generated in the compilation process. This includes executable files and object files. Excluding resources, if your final submission is bigger than 100Kb, you are probably including something unnecessary.

Trial Submission: Because we are using many different platforms, if you are not using the most traditional platform (Windows PC with Visual Studio C++ 2010 or higher), I would recommend that you perform a test submission of a sample program at least three days before the final deadline. (This is especially true for Mac users.) I will ask the TA to compile early submissions, and get back to you if he experiences any issues. Your trial submission does not need to do anything interesting. For example, it could just compile correctly and bring up a blank graphics window when executed.

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 so poorly written and organized that the TA has difficulty ascertaining your program’s structure. Since many of you are not familiar with C++ programming, we will not deduct points for poor C++ programming style. But, try to do your best.

Tips:

Units: Decide which units you want to use in representing your world. For simple 2-dimensional projects list this, it is natural to simply use actual pixels as the unit of distance, but you may find other dimensions to be more convenient (e.g., where each grid square of the board has side length 1).

State: All moving objects (the bullet and the falling bubbles) are characterized by their current physical state. This consists of a point \( p \) indicating the position of the center of the object and a velocity vector \( v \) indicating its speed and direction. Update these quantities in your idle (or timer) callback, and use their values to determine where to position the objects in your display callback.

Updates: With each update cycle (e.g., glut idle event), update the state. Based on the amount of time \( \Delta t \) that has elapsed since the last update, the object position can be updated as \( p \leftarrow p + \Delta t \cdot v \). After updating positions, check for collisions. (If objects are moving super fast, it is theoretically possible for one object to pass through another without detecting a collision. Don’t worry about this, since our objects will not be moving that fast.)

Tracing the Bullet Path: I found that tracing the path of the bullet was an interesting geometric problem. The challenge is that you need to determine not only which bubble of the board it hits, but the grid cell that immediately proceeds the hit (since this is where the new bubble goes). I did this by writing a procedure that, given the bullet’s starting point and directional vector, would trace the path of the bullet through the grid, cell by cell. I kept track of the current cell and the previous cell. When the current cell coincides with an existing bubble, the previous cell is where I placed the new bubble. To determine which neighboring cell is next involves a simple ray-shooting query to determine which edge (north, south, east, or west) the bullet path exits through. Once I knew the bullet’s final destination, I animated its path by walking along its path until reaching this destination.
Practice Problems for the Midterm Exam

The midterm exam will be on **Tue, Oct 29** 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 `gluLookAt()`, in what direction is the up vector not allowed to point. Explain.

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

\[
\begin{array}{c}
\text{y} \\
\text{d} \\
\text{p} \\
\text{c} \\
\text{a} \\
\text{b} \\
\text{x}
\end{array}
\]

(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.)

(c) You are told that two vectors $\vec{u}$ and $\vec{v}$ are each of length 2. What is the relationship (if any) between the dot product $(\vec{u} \cdot \vec{v})$ and the cosine of the angle between $\vec{u}$ and $\vec{v}$?

(d) A user draws a triangle strip using `GL_TRIANGLES_STRIP` and gives $n$ vertices. As a function of $n$, how many triangles are produced? (Assume there are 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 rendered using `glutSolidSphere`. One sphere is a pure diffuse reflector and the other is a pure specular reflector. Assuming the Gouraud shading model (which OpenGL uses), which of the two would require higher accuracy (that is, a greater number of slices and stacks) to produce a realistic shading of the sphere? Explain briefly.

(g) What is the halfway vector and why is it relevant to computing specular reflection? (Answer in a couple of sentences.)
Problem 2. You have a drawing rectangle of width $w$ and height $h$ (see the figure below (a)), which is to be mapped to a window of width $W$ and height $H$. The drawing rectangle is to be scaled uniformly so it fits entirely within the window, and (depending on the relationship of the two aspect ratios) either the width matches the width of the window or the height matches the height of the drawing window. In either case, it is centered within the window (see the figure below (b) and (c)). Given the glViewport command needed to map the graphics to the desired window. Explain how you derived your answer.

Problem 3. 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).)

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(int n, int 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: First determine how far it takes to perform a full $360°$ rotation.)

Problem 4. The folks in the art department came up with a function `drawBlock()`, which draws a 3-dimensional, axis-parallel block whose base is centered at the origin and whose side lengths are $2 \times 2 \times 8$ (see the figure below (a)).
(a) Your boss has decided that he wants to move the block so the center top is at the point \( p = (p_x, p_y, p_z) \) (see the figure (b)). Also, he wants to see the arrow on the right side, so he asks you to rotate it counterclockwise by 90° about vertical. Finally, he wants the dimensions changed to \( 6 \times 2 \times 4 \), so that the side with arrow remains of width 2. Explain how to use `drawBlock`, together with the OpenGL matrix operations, to achieve the desired result. (You cannot modify the function.) The matrix stack should be unchanged afterwards.

(b) Your boss wants to see what the original block looks like if it was tipped over, so that it falls 18° backwards towards the \(-x\)-axis. (see the figure (c)). Show how to use the `drawBlock` function to achieve this. (Be careful! The block rotates about its back edge, not about the origin.) As before, the matrix stack should be unchanged afterwards.

**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 \( \theta \). 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 6.** An anisotropic surface is one in which the shading varies depends on the direction to the viewer. The next version of OpenGL, called AwesomeGL, will support a new type of anisotropic surface material. Two RGB colors, \( C_0 \) and \( C_1 \), are given, as is a directional vector \( \vec{w} \) that runs parallel to the surface. Consider a point \( p \) on the surface that you want to color. Let \( \vec{v} \) be the normalized vector from \( p \) to the viewer, and let \( v^\perp \) be the projection of \( \vec{v} \) onto the surface. If \( v^\perp \) is perfectly aligned with \( \vec{w} \) (the angle between them is 0° or 180°) the surface color is \( C_0 \) (see the figure below (a)). If the angle is 90°, the surface color \( C_1 \) (see the figure below (b)). Between these two extremes, the colors change smoothly from one to the other.

(a) The viewer is at point \( q \), and the surface normal is \( \vec{n} \). Explain how to compute \( \vec{v} \) and \( v^\perp \)?

(b) Explain how to modify the Phong lighting model to handle this new type of surface. (Note that only the surface color changes; after this the lighting model must be applied. You may assume a single light source.)

**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 \( n \) vertical stacks (along the \( z \)-axis) and \( m \) radial slices (around the \( z \)-axis). (For example, in the figure we have \( n = 3 \) and \( m = 12 \).) You can draw using either `GL_QUAD_STRIPes` (one per stack) or a number of `GL_POLYGONs`.

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.** 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 (Ledo’s?) 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 \( z = -1 \).

![Pizza Diagram](image)

Where is the center?

(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.** Fog is a relatively easy enhancement to a shader. Fog is defined by three parameters, `fogStart`, `fogEnd`, and the fog RGB color \( F \). Let \( C \) be the color returned by the Phong lighting computations (ignoring fog). Let us assume that object coordinates have been converted into the view frame, so that given a point \( p \), its distance from the viewer is its distance from the origin, which is just the length of the vector \((p_x, p_y, p_z)\). If this distance is less than `fogStart` then \( C \) is used. If this distance is greater than `fogEnd` then \( F \) is returned. Otherwise, an appropriate mixture of the two colors is returned.

Give pseudocode for a function `getFog`, which returns the fog color, given the following parameters: the surface point \( p = (p_x, p_y, p_z) \), the natural surface color \( C = (C_R, C_G, C_B) \) returned from the Phong lighting computations, the fog distance parameters `fogStart`, `fogEnd`, and the fog color \( F = (F_R, F_G, F_B) \).
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. (30 points; 5–10 points for each part) Short answer questions. Except where noted, explanations are not required, but may be given for partial credit.

(a) For each of the following types of transformations, indicate which of the following properties holds. (For each transformation type, indicate all properties that apply.)

- Translation: ____________  (i) Maps lines to lines
- Rotation: ____________  (ii) Preserves midpoints
- Shearing: ____________  (iii) Preserves distances
- Perspective Projection: ____________  (iv) Preserves angles

(b) Recall that gluLookAt defines the view frame (also called the camera frame). The z-axis of this frame points in the opposite direction from the viewing direction. Why did the designers of OpenGL make this choice?

(c) In OpenGL, what is the purpose of mipmapping? (Select one.)

- (i) It is used to efficiently transfer bitmapped images from the CPU to the GPU.
- (ii) It is used to determine which object of a 3-dimensional scene is pointed to by the cursor.
- (iii) It is used in texture mapping to reduce the effects of aliasing.
- (iv) None of the above. (Explain what it is.)

(d) Consider the triangle strip shown in the figure below. Given the drawing order \((v_1, v_2, \ldots, v_7)\), which of the following statements is true? (Select one.)

\[
\begin{array}{c}
\text{\(v_1\)} \\
\text{\(v_2\)} \\
\text{\(v_3\)} \\
\text{\(v_4\)} \\
\text{\(v_5\)} \\
\text{\(v_6\)} \\
\text{\(v_7\)} \\
\end{array}
\]

- (i) The triangles of the strip are all front-facing.
- (ii) The triangles of the strip are all back-facing.
- (iii) They alternate between front and back.
- (iv) None of the above. (Explain.)

(e) In OpenGL lighting, the diffuse contribution to shading is proportional to \(\max(0, \vec{n} \cdot \vec{\ell})\) where \(\vec{n}\) is the surface normal and \(\vec{\ell}\) is the light vector. We were careful to add the “max” operator to avoid negative values. Under what circumstances would the result of \(\vec{n} \cdot \vec{\ell}\) be negative? (Select all that apply.)

- (i) The surface is facing away from the light source.
- (ii) The surface is invisible because it is behind the viewer.
(iii) A directional light sources (at infinity) is attenuated.

(iv) The value should never be negative. It was done to avoid taking square roots of negative numbers.

**Problem 2.** (25 points) You have been asked to produce some “word art”. You are given a function `draw(c)`, which draws character *c* so the bottom and left sides of its bounding rectangle are aligned with the *x*- and *y*-axes, respectively. (The figure below (a) shows the result of the call `draw('L')`.)

You also are given two functions `height(c)` and `width(c)`, which return the height and width of *c*, respectively. Each character has an implicit *reference point*, which lies at the midpoint of the bottom edge of the character’s bounding rectangle.

(a) Present a function `drawAt(char c, float x, float y, float theta)` which is given a character *c*, a point (*x*, *y*), and an angle *θ* given in degrees. This function should draw the character *c* rotated counterclockwise by angle *θ* with its reference point is located at (*x*, *y*) (see the figure above (b)). You can invoke the `draw` function, but you cannot modify it.

(b) Present a function `wordArt(char s[], float x, float y, float r)` which is given a character string *s*, a point (*x*, *y*), and a scalar *r*. This function draws the characters so that their reference points are evenly spaced along the upper half of a circle of radius *r* centered at the point (*x*, *y*), ranging from nine o’clock on the circle to three o’clock. You may assume that the string has at least two characters.

On return, the matrix stack should be *unchanged*. You may use the function `drawAt()` from part (a). Recall the system function `strlen(s)`, which returns the number of characters of string *s*. You can index the characters of *s* using *s[i]*.

**Problem 3.** (25 points) You have been assigned to work on a computer billiards game. You are given two points *p* and *q*, both with positive coordinates. A player must shoot a ball at point *p* = (*p_*x*, *p_*y*) that bounces off two walls to strike a second ball at point *q* = (*q_*x*, *q_*y*) (see the figure below (a)). The ball first hits the bottom wall (along the *x*-axis), next reflects to the side wall (along the *y*-axis), and finally reflects to hit *q*.

The objective of this problem is to compute the point *r* on the *x*-axis that is the first point of this double reflection (see the figure below (b)).
(a) Given the coordinates of \( p \) and \( q \), show how to compute a point \( q' = (q'_x, q'_y) \) that lies below the \( x \)-axis so that the line segment \( pq' \) intersects the \( x \)-axis at the point \( r \) (see the figure above (c)).

(b) Using the result from part (a), show how to compute the coordinates of the reflection point \( r \). Express your answer as a function of \( p \) and \( q \). (If you did not get part (a), you can still do this by expressing your answer in terms of \( p \) and \( q' \).

(c) Using the result from part (b), show that there exists a projective transformation \( T \) (depending on \( q \) but not on \( p \)) such that \( r = T(p) \). More formally, show that there exists a \( 3 \times 3 \) matrix \( T \) such that, if \( r \) and \( p \) are expressed in projective homogeneous coordinates, then after perspective normalization, we have \( r = T \cdot p \). That is,

\[
T \cdot p = \begin{bmatrix} t_{1,1} & t_{1,2} & t_{1,3} \\ t_{2,1} & t_{2,2} & t_{2,3} \\ t_{3,1} & t_{3,2} & t_{3,3} \end{bmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix} = \begin{pmatrix} r'_x \\ r'_y \\ r'_w \end{pmatrix} \equiv \begin{pmatrix} r'_x/r'_w \\ r'_y/r'_w \\ 1 \end{pmatrix} = \begin{pmatrix} r_x \\ r_y \\ 1 \end{pmatrix}
\]

**Problem 4.** (20 points) Give a function that draws a cylinder centered on the \( z \)-axis (see the figure below (a)). Let \( h \) and \( r \) denote the cylinder’s height and radius, respectively, and let \( m \) denote the number of radial slices. The cylinder’s bottom lies on the \( x, y \)-plane (\( z = 0 \)). Additional requirements:

- The top and bottom should be rendered as a GL_TRIANGLES_FAN’s centered on the \( z \)-axis (see the figure above (b)). The sides should be drawn as a single GL_QUAD_STRIP.
- Specify both vertices and surface normals. The top and bottom normals are directed up and down, respectively. The side normals are perpendicular to the \( z \)-axis (see the figure above (c)).
- All faces should have a counterclockwise orientation from outside the cylinder.

Present a function `drawCylinder(float h, float r, int m)` that draws the cylinder as described above.
Programming Assignment 2: Sky Chess (Phase I)

Handed out Thu, Nov 7. The first phase must be submitted through the submit server by Mon, Nov 25, 11:59pm. See the syllabus for the late policy.

Overview. The ultimate goal of this project is to implement the front-end for a 3-dimensional chess program. Your program will provide the basic elements of the user-experience. In particular, this will involve displaying a perspective view of the chess board and its pieces, allowing the camera to swivel around the board and to zoom-in and out. It will also allow the user to select pieces in order to move them from one position to another. In the final version, your program will animate the movement of the pieces. The basic elements you will learn include the following:

- Processing both keyboard and mouse inputs (Phase I)
- Allowing the user to control the camera position (Phase I)
- Rendering and manipulating mesh models (Phase I)
- Added realism through lighting and texture mapping (Phases I and II)
- Selecting objects of the scene by mouse input (Phase II)
- Added realism through shadows (Phase II)
- Simple animation (Phase II)

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

Phase II: Add picking, multiple textures, shadows, animation.

Phase II will be described later in greater detail. 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.)

World: (Required) The game takes place on a chessboard platform (called the board) that floats in space. The pieces sit on the board. (In the first phase they will be centered within individual squares of the board, but in Phase II, the pieces will move smoothly around the board, will fly through space, and will generally rotate, so plan accordingly.) The board is enclosed within a much larger enclosure, which is texture mapped with a skybox.

Camera motion: (Required) The camera’s elevation and distance can be adjusted by the user. The camera is positioned at a fixed set of spherical coordinates relative to the center point of board. When the mouse is dragged up and down, the camera’s elevation increases and decreases. When the mouse is dragged left and right, the camera rotates horizontally around a vertical line passing through the center of board. Through the use of two key inputs (we used ‘o’ and ‘i’) the camera can be zoomed out or zoomed in. You should avoid allowing the elevation to grow too high or too low. (We limited ours to about 85° above and below.)
Texture mapping: (Required) Your program must have at least one element of texture mapping, namely a texture-mapped sky-box, that surrounds the board, creating the impression that the board is floating in space. (Note that multiple textured objects will be required for Phase II, so you may want to plan accordingly. In particular, you will be asked to apply texture-mapping to the board.)

Lighting: (Required) Your program should make use of at least one light source to illuminate elements of your scene. (In Phase II, you will be required to generate board shadows for one of your light sources.)

Modeling: (20%) Although you may design your own pieces, you must support at least one piece shape (a pawn) that we will provide to you. This will be a silhouette-based model (as in the Challenge Problem on Homework 1), which you will map into a surface of revolution and render as a mesh (e.g., using some combination of OpenGL triangle fans, triangle strips, and/or quad strips).

Cursor-based Selection: (This requirement is being moved to Phase II. You may implement as part of Phase I for extra credit.) Your program should allow the user to select positions on the chessboard through mouse input. (This process is called picking.) In class, we will discuss a simple and efficient method for picking that is based on ray-tracing. (It is not required that you use this method, but since ray-tracing is an important technique in computer graphics, I would encourage you to use this method.)

Resources: We will make a sample executable available on the class projects page (under “Projects”), some simple software for inputting texture images, and some other useful files (including the chess-piece model and texture maps).

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., software for performing basic geometry and linear algebra, software for loading textures, and code-snippets for minor technical tasks). If you are not sure whether some code satisfies our conditions of “fair use”, please check with me.
Programming Assignment 2: Sky Chess (Phase II)

Handed out Tue, Dec 3. Due (through the submit server) by Wed, Dec 11, 11:59pm. See the syllabus for the late policy.

Overview. This is the second phase of the Sky Chess project. Recall the overall goals of the project:

- Processing both keyboard and mouse inputs (Phase I)
- Allowing the user to control the camera position (Phase I)
- Rendering and manipulating mesh models (Phase I)
- Added realism through lighting and texture mapping (Phases I and II)
- Selecting objects of the scene by mouse input (Phase II)
- Added realism through shadows (Phase II)
- Simple animation (Phase II)

Phase I requirements: Because many key functional aspects of Phase I are needed in this phase as well, we will expect you to fix any unimplemented elements of Phase I that are required in Phase II. This includes mouse/keyboard-based camera control, rendering mesh models, lighting and texture mapping.

Picking: When the mouse is moving about (without the user holding any buttons down) the chess-board square under the cursor should be highlighted in some manner. (For example, you can change its color or texture.) When the mouse button is clicked on a square that contains a piece, the square should be permanently highlighted to indicate that this piece has been selected. (I used the right button in my implementation, since the left button was used for camera control.)

Multiple Texture Maps: In Phase I you texture mapped the skybox. In this phase you must apply texture mapping to some other objects. In my implementation, I texture mapped the squares of the board using a wood-grain texture (which was provided with the resource file).

Shadows: (20%) The chess pieces should cast shadows in a realistic manner. At a minimum the shadows should be cast onto the chess board. (This can be done using the shadow painting method discussed in class.) If you wish to experiment with more complex methods, you can try casting shadows from the pieces onto other pieces. (The shadow map method would be appropriate for doing this.)

Animation of Pieces: (25%) When a chess piece is selected, it should initiate some simple animation to indicate that this piece has been selected. (I had the piece jump up and down.) When another square is selected, the piece should move smoothly to that other square. (I had the piece slide along the board, but you may handle this any way you like, provided that the movement looks smooth and natural.) Finally, if the moved piece lands on a square occupied by another piece, this other piece should be knocked from the board through a smooth animation. (I have not implemented this yet, but my plan was that the piece would appear to be blown off the board as if an explosion took place underneath it.) There is no requirement that any of the rules of chess be observed when moving pieces.

External Resources: The same rules about the use of external resources as last time applies. You are allowed to use code pieces (e.g., if you find code for picking or shadows), but you are responsible for understanding how this code works and you are required to cite any such resources in your ReadMe.txt file. If you are not sure whether some code satisfies our conditions of “fair use”, please check with me.
Practice Problems for the Final Exam

The final will be on **Mon, Dec 16, 8:00–10:00am**. The exam will be closed-books, closed-notes, but you will be allowed two sheets of notes, front and back to use for the exam. 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) Given two vectors $\vec{u}$ and $\vec{v}$ in 3-dimensional space, you are told that $\vec{u} \times \vec{v} = 0$. Which of the following can you infer from this? (List all that apply.)

(i) One or both vectors must be the zero vector, $(0, 0, 0)$.

(ii) If the vectors are both nonzero, they are perpendicular to each other.

(iii) If the vectors are both nonzero, they are parallel to each other.

(iv) The dot product $(\vec{u} \cdot \vec{v})$ must also be zero.

(b) Answer the following questions involving the function `gluPerspective(fovy, aspect, near, far)`.

(i) If the ratio $\text{far}/\text{near}$ is made unnecessarily large, what sort of error could arise in the final image?

(ii) Suppose you wanted to produce an effect of zooming in to produce a close-up of an object at the center of the image. Which parameter would you change? Would you decrease or increase its value?

(iii) Suppose that you have a viewport whose lower left corner is at $(5, 10)$ and whose width and height are 20 and 50, respectively. What would you set the aspect parameter to be (assuming you want no distortion)?

(c) 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

(d) The cross product of two unit vectors $\vec{u} \times \vec{v}$ is of unit length. What can be said about their dot product, $(\vec{u} \cdot \vec{v})$? (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.

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

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

Recall that $\vec{n}$ 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) Explain the difference in how smooth shading is performed in Phong shading and Gouraud shading. Which method does OpenGL use?

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

(h) A ray is shot at a transmissive and nonreflective surface, and total internal reflection occurs. From which side did the ray strike? (i) the one of higher IOR (index of refraction), (ii) the one of lower IOR, or (iii) could be either.

(i) In ray tracing, whenever a ray arrived at a surface, we shot rays to each of the light sources. What was the purpose for doing this?

(j) Give the three blending functions for a Bézier curve of order 2.

(k) B-splines possess a property called local support. What is local support, and why is this property desirable?

Problem 2. An important utility is drawing text strings. This question will consider doing this, assuming you have access to a function that draws individual characters. Assume that each character is defined by an integer code (e.g., its ASCII or Unicode value). You are given the following:

- The function `draw(i)` draws the character whose character code is `i`, so that its lower left corner is at the origin. (See the figure below (a).)
- Different characters have different widths. You are given an array of widths, where `width[i]` holds the width of the `i`th character.
- You are given a character string to draw, where `string[j]` contains the character code of the `j`-th character to be drawn, and `string.length` is the number of characters in the string.

```
width('L')
draw('L')
```

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

(b) Implement a procedure `drawString2(x0, y0, s, θ, string)` that draws the given string so that its lower left corner is at the point `(x0, y0)`, it is scaled uniformly by the factor `s`, and rotated by angle `θ`, given in degrees (see figure (c)). Except for the string, all the arguments are of type `GLfloat`. (Hint: You may call your procedure from part (a).)
Problem 3. 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 4. Consider the cone shown in the figure below. 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, show that any point $p = (x, y, z)$ on the cone can be expressed as $x(u, v), y(u, v)$ and $z(u, v)$, for some choice of reals $u$ and $v$.

(b) Given your answer to (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. 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.** 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 6](image)

**Problem 7.** You have been asked you to produce a ray-intersection procedure for cereal-bowl shape. 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 $\vec{u}$ be a unit vector. Given a ray $p + t\vec{u}$, 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 objects with a reflective surface in OpenGL. First, you have a table, which is a horizontal (filled) rectangle that floats above the ground at height $z = 10$. The function $\text{drawTable}()$ draws the table. On the table is a puddle of spilled water, which behaves like a perfect reflector. It is drawn using the function $\text{drawPuddle}()$ (see part (a) of the figure below). Finally, there are some objects that sit on the table (a sphere and a pyramid shown in part (b) of the figure below), which are drawn by a function $\text{drawObjects}()$.

![Problem 8](image)

(a) Give a function $\text{drawReflectedObjects}()$, which invokes $\text{drawObjects}()$ so that the objects are drawn as if they have been reflected below the table top (see part (c) of the figure). The matrix state should be unchanged on returning from this function.
(b) Give a function to render the entire scene: table top, objects, and puddle with reflected objects. You are not required to give specific OpenGL commands, but it should be clear how to translate each of your operations directly into OpenGL (e.g., “save the matrix state”, “disable the depth test”, “draw a given shape into the stencil buffer with reference value . . . ”, 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.

You may ignore lighting issues throughout.

**Problem 9.** Let \( b_{k,d}(u) \) denote the \( k \)-th B´ezier blending function of degree \( d \). Recall that given an array \( p_{ij} \), of control points \( 0 \leq i, j \leq 3 \), the B´ezier surface is given by

\[
p(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_{i,3}(v) b_{j,3}(u) p_{ij},
\]

for \( 0 \leq u, v \leq 1 \). Given any fixed \( u_0, 0 \leq u_0 \leq 1 \), define the \( u_0 \)-slice to be the curve \( s(v) = p(u_0,v) \). Show that \( s(v) \) is a B´ezier curve of degree 3. What are the four control points for this curve?

**Problem 10.** Given three control points \( p_0, p_1, \) and \( p_2 \) in the plane, recall that the B´ezier curve of degree-2 is:

\[
B(u) = (1-u)^2 p_0 + 2u(1-u)p_1 + u^2 p_2.
\]

Prove the following claim: For any three points \( p_0, p_1, \) and \( p_2 \), the tangent to the curve at \( u = 1/2 \) is parallel to the line segment \( p_0 p_2 \) (see the figure below). Prove this from first principles.

(Hint: Start by computing the derivative of \( B(u) \), carefully.)
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. (30 points; 2–8 points each) Short answer questions. (Explanations are not required, but may be given for partial credit.)

(a) Given two vectors \( \vec{u} \) and \( \vec{v} \) in 3-dimensional space, you are told that their dot product is zero, that is, \( (\vec{u} \cdot \vec{v}) = 0 \). Which of the following necessarily follows as a logical consequence of this? (List all that apply.)

(i) One or both vectors must be the zero vector, \((0, 0, 0)\).
(ii) If the vectors are both nonzero, they are perpendicular to each other.
(iii) If the vectors are both nonzero, they are parallel to each other.

(b) Suppose you want to draw a scene using \texttt{gluPerspective} that is free from distortion (in the sense that objects are neither stretched out horizontally or vertically). Which of the following methods achieves this. Assume that your viewport is not equal to the entire window. (List all that apply.)

(i) The fovy given to \texttt{gluPerspective} must be equal to the height of your viewport.
(ii) The aspect-ratio given to \texttt{gluPerspective} must equal the aspect ratio of your graphics window.
(iii) The aspect-ratio given to \texttt{gluPerspective} must equal the aspect ratio of your viewport.
(iv) The ratio of the distances to the far and near clipping planes should not exceed \(2^b\), where \(b\) is the number of bits in your depth buffer.

(c) Indicate which of the three shadow-generation methods achieves this property. Possible choices are \textit{shadow painting}, \textit{shadow mapping}, and \textit{shadow volumes}. (List all that apply.)

(i) This method can efficiently handle complex occluding surfaces (that is, the surfaces that cast shadows).
(ii) This method can efficiently handle complex surfaces onto which the shadows are projected.
(iii) This method relies primarily on the depth buffer to determine which fragments are in shadow and which are not.
(iv) This method relies primarily on the stencil buffer to determine which fragments are in shadow and which are not.

(d) You are given an implicit surface in 3-dimensional space, represented by the equation \( f(x, y, z) = 0 \). Explain how to compute a surface normal at a particular point \((x_0, y_0, z_0)\) that lies on this surface.

(e) When talking about curves and surfaces defined by control points, what is the meaning of the term \textit{local support} and why is it desirable? Between Bézier curves and B-spline curves, which (if any) have local support? (List all that apply.)
Problem 2. (10 points) Given three control points \( p_0, p_1, \) and \( p_2 \) in the plane, recall that the Bézier curve of degree-2 is:

\[
B(u) = (1 - u)^2 p_0 + 2u(1 - u)p_1 + u^2 p_2.
\]

Prove the following claim: For any three points \( p_0, p_1, \) and \( p_2, \) the tangent to the Bézier curve at \( u = 1 \) is parallel to the line segment \( p_1p_2 \) (see the figure below). Prove this from first principles. You may not use any facts about Bézier curves given in class.

Problem 3. (20 points) You are given a cylinder of radius 1 aligned with the \( z \)-axis, running from \( z = 0 \) to \( z = 2 \). You are given a texture map whose \( s \) and \( t \) coordinates range from 0 to 1. You are to wrap this texture around the cylinder so that the central third of the texture \((1/3 \leq t \leq 2/3)\) appears like a “candy-cane stripe.” The stripe should start above the \( x \)-axis, and wrap exactly once around the cylinder (see the figure below).

In this problem you will derive the inverse wrapping function, which maps a point \((x, y, z)\) on the cylinder to the corresponding point \((s, t)\) on the texture.

(a) Give a parametric representation of the cylinder. That is, given two real parameters \( u \) and \( v \), show that any point \( p = (x, y, z) \) that lies on the cylinder 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 cylinder?

(c) Give the inverse wrapping function, which maps a point \((x, y, z)\) on the cylinder to the corresponding point \((s, t)\) in texture space. (Don’t worry about the gray areas. I only care that your function works for the points on the stripe.)

Problem 4. (20 points) You have been asked you to produce a ray-intersection procedure for a cone shaped object. The cone’s axis is aligned with the \( z \)-axis, its apex is at the origin, and its base has radius 1 and is located at \( z = 2 \). The cone is hollow (like a cheap paper cup), meaning that it has no base surface.
(a) Give an implicit function representation of this cone, by giving a function \( f \) such that \( f(x, y, z) = 0 \) for each point on the surface of the (infinite) cone. (Hint: In order to answer (b), your function should be expressed as a polynomial of degree at most 2.)

(b) Let \( p \) be a point and \( \vec{u} \) be a unit vector. Given a ray \( p + t\vec{u} \), present a procedure (as either mathematical formulas or pseudo-code) that determines the value \( t \) of the first intersection of the ray with the cone. If there is no intersection with the cone, 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 \( (-b \pm \sqrt{b^2 - 4ac})/2a \).)

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 \([1, 3]\) along the \( x \)-axis and from \([1, 4]\) along the \( z \)-axis. Write an OpenGL procedure to render this scene and its reflection in the mirror through the use of the stencil buffer. You may assume that 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 polygon \( P \) into the stencil buffer setting pixel values to 1”, 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?
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 and Enumerations:** 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:

```cpp
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.

A common use for constants is for indicating quantities that take on a discrete set of values. As in C, C++ offers *enumerations*, which provide an easy way to define such constants. Here are some examples:

```cpp
enum DayOfWeek { SUN, MON, TUE, WED, THU, FRI, SAT }
enum ExecutionMode { DEBUG, PERF_TEST, RELEASE }
```

```cpp
DayOfWeek today = TUE;
ExecutionMode execMode = DEBUG;
```

**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.*)

```cpp
class Vector2d {
   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; }
```
We have defined the getters and setters inside the class. We have chosen to define the constructor and the addTo function outside the class (see below).

```cpp
#include "Vector2d.h"

Vector2d::Vector2d(double xx, double yy)
{ x = xx; y = yy; }

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

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? Some C++ purists insist that all member functions be defined outside the class (to prevent a user of the class from seeing them). Other C++ programmers define short 1-line member functions inside the class definitions (e.g., getters and setters) and define the rest externally. 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.

```cpp
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.

```cpp
#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 deallocate memory that has been allocated. Failure to deallocate memory that has been 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
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. By vector, I mean a variable-sized array (which may be appended to), and by list, I mean a doubly linked list. Rather than going through the hassles of defining your own vector and list structures, and dealing with the headaches of memory allocation and deallocation, it is much simpler to use the built-in vector and list types provided by the C++ Standard Template Library, or STL. The STL provides data structures for a number standard containers, such as stacks, queues, deques (double ended queues), vectors, lists, priority queues, and maps.

One of the important features of the STL is that each can store objects of any one type. Such a class whose definition depends on a user-specified type is called a template. The type of the object being stored in the container is given in angle brackets. For example, we could define vectors to hold 100 integers or 500 characters as follows:

```cpp
#include <vector> // class vector definitions
using namespace std; // make std accessible
vector<int> scores(100); // a vector of (initially) 100 integers
vector<char> buffer(500); // a vector of (initially) 500 characters
```

Here are a couple of examples of how to use an STL vector.

```cpp
int n = 100;
vecto<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 append 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.

STL vectors are superior to standard C++ arrays in many respects. First, as with arrays, individual elements can be indexed using the usual index operator ([ ]). They can also be accessed by the at member function, which also performs array bounds checking. (As in C, arrays in C++ do not even know their size, and hence range checking is not even possible.) In contrast, a vector object's size is given by its size member function. Unlike standard arrays, one vector object can be assigned to another, which results in the contents of one vector object being copied to the other. A vector can be resized dynamically by calling the resize member function. Here are some examples:

```cpp
int i = // ...
cout << scores[i]; // index (range unchecked)
buffer.at(i) = buffer.at(2 * i); // index (range checked)
vecto<int> newScores = scores; // copy scores to newScores
scores.resize(scores.size() + 10); // add room for 10 more elements
```

Another handy STL container is the list, which implements a doubly linked list. Here is how to declare a list of floats. By default, the initial list is empty.
List supports operations such as size (number of elements), empty (is the list empty?), front/back (return reference to first/last elements), push_front/push_back (insert to front/back), pop_front/pop_back (remove from front/back).

Iterators: The STL container classes introduced above all define a special associated class called an iterator. An iterator is an object that specifies a position within a container and which is endowed with the ability to navigate to other positions. If p is an iterator that refers to some position within a container, then *p yields a reference to the associated element.

Advancing to the next element of the container is done by incrementing the iterator. For example, either ++p or p++ advances p to point to the next element of the container. The former returns the updated value of the iterator, and the latter returns its original value. Each STL container class provides two member functions, begin and end, each of which returns an iterator for this container. The first returns an iterator that points to the first element of the container, and the second returns an iterator that can be thought of as pointing to an imaginary element just beyond the last element of the container.

Let us see how we can use iterators to enumerate the elements of an STL container C. Suppose, for example, that C is of type vector<int>, that is, it is an STL list of integers. The associated iterator type is denoted "vector<int>::iterator". For example, the code below demonstrates how to sum the elements of an STL vector V using an iterator.

```cpp
typef vector<int>::iterator Iterator; // iterator type
int sum = 0;
for (Iterator p = v.begin(); p != v.end(); ++p) {
    sum += *p;
}
```

Different containers provide iterators with different capabilities. Most STL containers (including lists, sets, and maps) provide the ability to move not only forward, but backward as well. For such containers the decrement operators --p and p-- are also defined for their iterators. This is called a bidirectional iterator.

A few STL containers (including vectors and deques) support the additional feature of allowing the addition and subtraction of an integer. For example, for such an iterator, p, the value p + 3 references the element three positions after p in the container. This is called a random-access iterator. As with pointers, care is needed in the use of iterators. For example, it is up to the programmer to be sure that an iterator points to a valid element of the container before attempting to dereference it. Attempting to dereference an invalid iterator can result in your program aborting. As mentioned earlier, iterators can be invalid for various reasons. For example, an iterator becomes invalid if the position that it refers to is deleted.

References: In C, all parameter passing to functions is performed by value. This has two important implications. First, altering the value of a 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 \( i \).

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

References are rarely used as shown above. Instead, they are principally used for passing parameters, “by reference” to functions. A reference parameter has two advantages over a value parameter. First, it can be modified (without any messy pointer dereferencing) and it is more efficient for class objects, since only the address of the object (not the entire object) needs to be conveyed to the function.

```cpp
void f(int& r, Vector2d& u) {
    r = 27; // changes the actual parameter to 27
    u.setY(4); // alters y-coordinate of actual parameter
}
```

// ... in your main program
```cpp
int i = 34;
Vector2d v(2, 1);
f(i, v);
```
```cpp
cout << i << "  " << v.getY() << "\n"; // outputs "27 4"
```

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:

```cpp
Vector2d add(const Vector2d& u, const Vector2d& v) {
    // ... in this function u and v may be read, but not modified
}
```

**More information:** There are a number of resources on the web, which can be found by searching for “C++ for Java programmers”. Here are some examples:

- [http://www.cs.williams.edu/~lenhart/courses/cs371/c++forjava.html](http://www.cs.williams.edu/~lenhart/courses/cs371/c++forjava.html)