CMSC 427: Computer Graphics

Spring 2004

http://www.cs.umd.edu/~mount/427/

Instructor: Dave Mount. Office: AVW 3373. Email: mount@cs.umd.edu. Office phone: (301) 405–2704. Office hours: Mon 2:30-3:30, Wed 2:30-3:30. I am also available immediately after class for questions. Please send me email if you cannot make these times. If the question is short (a minute or so) drop by my office any time. Please send me email if you cannot make these times. (Don't be shy about doing this. I always set aside at least one hour each week for "unscheduled" office hours.)

Teaching Assistant: Pooja Nath. Office: AVW 1112. (If you do not see her there, try her office, AVW 3444.) Email: pooja@cs.umd.edu. Office hours: (TBA, see the class web page). If you cannot make these times, please feel free to contact her to set up another time.

Class Time: Tue, Thu 2:00-3:15 in CSI 3117.

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

Texts: It is strongly recommended that you buy the required text. The reference book is a good source of advanced information, if you intend to do advanced graphics programming.

Required: *Computer Graphics with OpenGL* (3rd edition), D. Hearn and M. P. Baker, Prentice Hall, 2004.

Reference:

- *OpenGL Programming Guide: The Official Guide to Learning OpenGL* (Fourth Edition), by OpenGL Architecture Review Board, *et al.*, Addison-Wesley, 2003.
- *OpenGL Reference Manual: The Official Reference Document to OpenGL* (3rd Edition), by Dave Shreiner, *et al.*, Addison-Wesley, 1999.

Prerequisites: MATH 240 (Linear Algebra) and CMSC 420 (Data Structures). Knowledge of C, C++, or Java programming. The course involves a considerable amount of mathematical reasoning involving 3-dimensional objects (points, lines, spheres, and polygons). Knowledge of and the ability to solve problems in linear algebra and (primarily differential) calculus will be required. We will give an overview of linear algebra in class, but if you are unfamiliar in your understanding of concepts from linear algebra (such as vector spaces, linear independence, bases, linear transformations, determinants, and inner products) and differential calculus (including partial derivatives), I recommend that you review this material. The course involves some nontrivial programming projects. Although a specific knowledge of data structures is not essential, I will assume that you are capable of writing and debugging moderately sophisticated programs in either C, C++, or Java.

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

There will be two exams: a midterm and a comprehensive final. Tentative weights: Homeworks and projects 35%, midterm 25%, final exam 40%. The final exam will be Mon, May 17, 10:30–12:30.

As a courtesy to the grader, homework assignments are to be written up neatly and clearly, and programming assignments must be clear and well-documented. Although you may develop your program on whatever system you like, for final grading your program must execute either on a workstation (Microsoft Windows, Linux, or Sun Solaris) either in the WAM, Glue, or Linux labs. If you develop your program on some other platform, it is your responsibility to see that it can be compiled and executed on one of these machines by the due date. If not, you will be asked to make whatever changes are needed and will be assessed a late penalty as a result.

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

Academic Dishonesty: All class work is to be done independently. You are allowed to discuss class material, homework problems, and general solution strategies with your classmates. When it comes to formulating/writing/programming solutions you must work alone. If you make use of other sources in coming up with your answers you *must* cite these sources clearly (papers or books in the literature, friends or classmates, information downloaded from the web, whatever).

It is best to try to solve problems on your own, since problem solving is an important component of the course. But I will not deduct points if you make use of outside help, provided that you cite your sources clearly. Representing other people's work as your own, however, is plagiarism and is in violation of university policies. Instances of academic dishonesty will be dealt with harshly, and usually result in a hearing in front of a student honor council, and a grade of XF.

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

Introduction: Overview of graphics systems, graphics devices, graphics programming.

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

Geometric Programming: Review of linear algebra, affine geometry, (points, vectors, affine transformations), homogeneous coordinates, change of coordinate systems.

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

Light and shading: Diffuse and specular reflection, Phong and Gouraud shading.

Using Images: Texture-, bump-, and reflection-mapping.

Implementation Issues: Rasterization, clipping.

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

Hidden surface removal: Back-face culling, z-buffer method, depth-sort.

Color: Gamma-correction, halftoning, and color models.

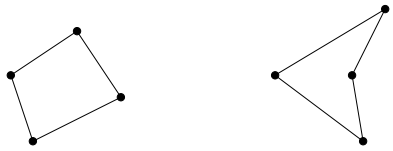
Modeling: Hierarchical models, fractals and fractal dimension.

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

Homework 1: OpenGL and Geometry

Handed out Tuesday, Feb 24. Due at the start of class Tuesday, Mar 2. Late homeworks are not accepted, so turn in whatever you have done.

Problem 1. Suppose that you are given the vertices of a 4-sided polygon, $\langle P_0, P_1, P_2, P_3 \rangle$ in the plane. Recall that a polygon is *convex* if every internal angle is less than or equal to 180 degrees.



(a) Assuming that these vertices have been given in counterclockwise order, explain how to use orientation tests to determine whether this polygon is convex.

(b) Repeat (a), but now the ordering of the vertices is unknown, but it will be either clockwise or counterclockwise.

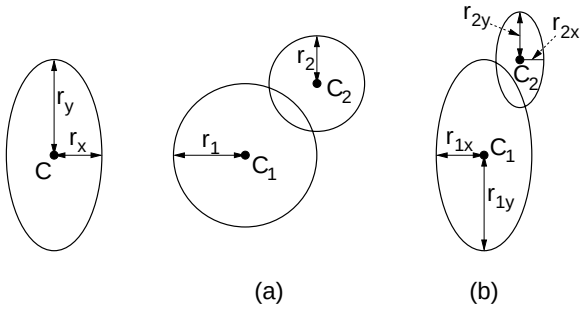
Problem 2. Given a point $P = (p_x, p_y, 1)^T$ in the plane and an angle θ, derive a transformation that rotates the plane by θ degrees clockwise (not counterclockwise) about the point P. As we did in class, express your answer as a 3×3 matrix, so that it can be applied to a column vector in homogeneous coordinates. Show how you derived your answer.

Problem 3. For this problem you "might" need to use the following facts (depending on how you solve the problem). An axis-aligned ellipse in the plane centered at a point $C = (c_x, c_y)$ and with horizontal and vertical radii $r_x > 0$ and $r_y > 0$ satisfies the equation

$$\frac{(x - c_x)^2}{r_x^2} + \frac{(y - c_y)^2}{r_y^2} = 1.$$

Define the *aspect ratio* of the ellipse to be r_y/r_x. In the case where $r_x = r_y = r$ (aspect ratio is 1) the ellipse is a circle of radius r, in which the above equation is equivalent to $(x - c_x)^2 + (y - c_y)^2 = r^2$. Answer each of the following questions.

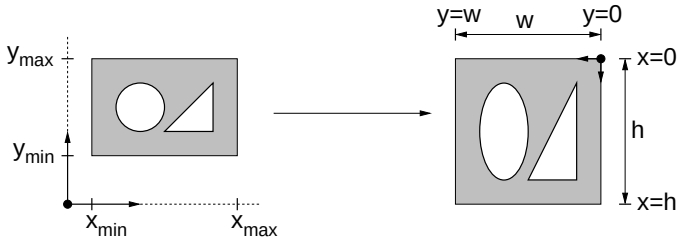
(a) You are given two circles in the plane of radii r_1 and r_2 centered at points $C_1 = (c_{1x}, c_{1y})$ and $C_2 = (c_{2x}, c_{2y})$, respectively. Derive an expression that tests whether these two circles overlap each other, but one circle is not contained within the other. (See the figure below.)



(b) You are given two axis aligned ellipses, centered at points C_1 and C_2, with horizontal and vertical radii r_{1x} and r_{1y} and r_{2x} and r_{2y}, respectively. Further, assume that the two ellipses have the same aspect ratios. Repeat part (a) in this case.

(Hint: The easy way to solve both problems does not involve the use of the above equations for the circle and ellipse.)

Problem 4. On the distant planet of Omicron Persei 8 (OP8), graphics viewports are designed so that the viewport origin is in the upper right corner. The x-axis points down and the y-axis points to the left. Let w and h denote the height and width of the OP8 viewport. (See the figure below.) Consider a rectangular drawing region (using the standard coordinate system) whose left and right sides are $x_{\min}$ and $x_{\max}$ and whose bottom and top sides are $y_{\min}$ and $y_{\max}$.



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

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

Show how you derived your answer.

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

$$P = \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix} \qquad V = \begin{pmatrix} v_x \\ v_y \\ 1 \end{pmatrix}.$$

Express the transformation of part (a) as a 3×3 transformation matrix M, so that

$$V = MP.$$

(Optional) Homework 2: Perspective and Lighting

This is an optional homework. This means that the homework will not be considered as part of your total numerical grade, but instead will be assigned to extra credit points. (See the syllabus for more information on extra credit.) The homework also serves as a set of sample problems for the midterm exam.

Handed out Thursday, Mar 18. Due at the start of class Thursday, April 1. Late homeworks are not accepted, so turn in whatever you have done.

Problem 1. Short answer questions. Explanations are optional, but may be provided for partial credit.

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

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

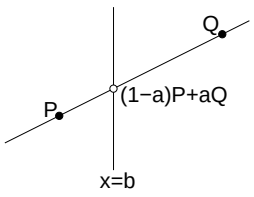
(c) A user draws a triangle strip using GL_TRIANGLE_STRIP and gives n vertices. As a function of n, how many triangles are produced? (Assuming no three collinear vertices and no duplicate vertices.)

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

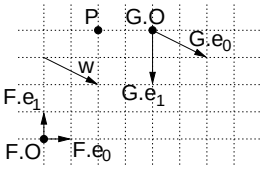
(e) Which of the following statements is true of perspective projections? (Select all that apply)

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

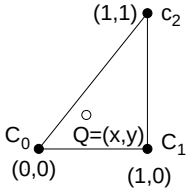
(f) Given points P_0, P_1, P_2 in 3-space, and a viewer at point V, give a geometric test to determine whether, from V's location, the vertices of triangle $\triangle P_0P_1P_2$ appear in clockwise or counterclockwise order.



Problem 1(d)



Problem 2



Problem 3

Problem 2. Consider the two frames F and G shown in the figure above.

(a) Express both P and $\vec{w}$ in homogeneous coordinates relative to frame F.

(b) Express both P and $\vec{w}$ in homogeneous coordinates relative to frame G.

(c) Give the 3×3 matrix which transforms a point represented in homogeneous coordinates relative to G into its homogeneous coordinates relative to F. (If you wish, you may express your answer as the inverse of a matrix, without actually computing the inverse.)

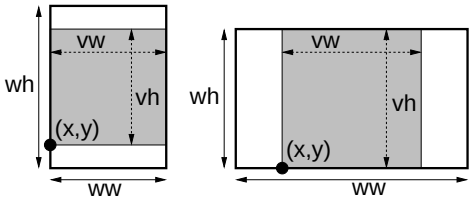
Problem 3. As mentioned in class, Gouraud shading is performed by linearly interpolating the colors of the vertices of a polygon to the points in the interior of the polygon. Consider the triangle in the figure above, with vertices at $(0, 0)$, $(1, 0)$ and $(1, 1)$. Let C_0, C_1 and C_2 denote the corresponding RGB color vectors assigned to these three vertices.

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

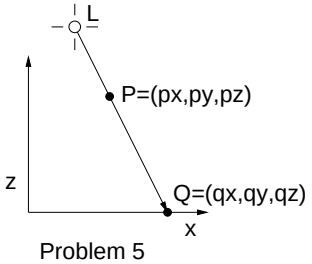
Problem 4. Suppose that you have a graphics window that must be a square. The user has just resized the window so that it now has width ww and height wh. As a function of ww and wh, derive the arguments for glViewport() so that the new viewport is the largest square that fits within the window and is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

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

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



Problem 4



Problem 5

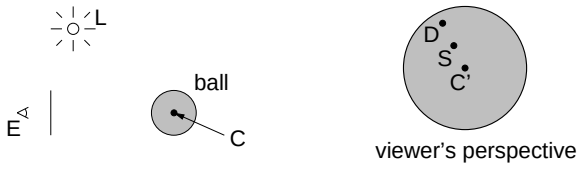
Problem 5. OpenGL does not compute shadows. One way to produce the shadow of an object is to explicitly compute them yourself and just draw the shadows. Let us consider how to do derive a function to do this. Let $L = (\ell_x, \ell_y, \ell_z)^T$ be the coordinates of a light source and let $P = (p_x,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

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

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

Challenge Problem. A viewer is looking at a spherical ball centered at a point C in 3-space that has both diffuse and specular reflections. The ball is illuminated by a single point light source L. The viewer at location E takes a picture of the scene. He observes two points on the ball in his image: D is the point of the brightest diffuse reflection and S is the point of the brightest specular reflection. Let C' be the location of the center point of the ball from the viewer's perspective. (See the figure below.) You are not told the exact location of the viewer or the light source, but you may assume that both are far away from the ball, that D is on the side of the ball that is visible to the viewer, and that L, C, and E are not collinear.

Prove that, from the viewer's perspective the points D, S, and C are collinear? In the figure we show that S lies between D and C. Explain why this is so. (Hint: Try to reduce this to a two dimensional problem, by considering an appropriate plane.)



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!

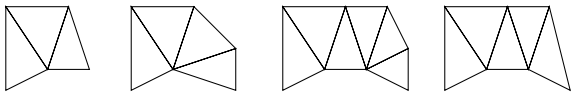
Problem 1. (30 points; 3–6 points each) Short answer questions. Explanations are not required, but may be given for partial credit

(a) In the call

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

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

(b) Consider the groups of triangles shown below. For each one, indicate whether it can be drawn as (1) a single triangle strip, (2) a single triangle fan, (3) could be drawn as either, or (4) cannot be drawn as a single triangle strip or a single triangle fan. No further explanation required. (Note that each triangle must be drawn as shown in the figure, and you cannot draw empty triangles.)



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

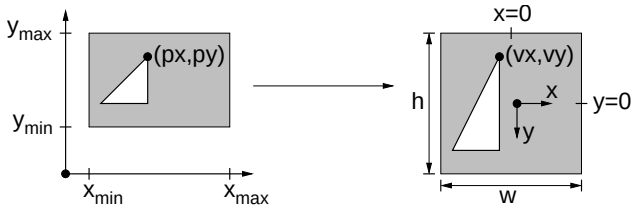
(d) Given two nonzero vectors $\vec{u}$ and $\vec{v}$ in 3-dimensional space, the operation $\vec{u} \times \vec{v}$ will produce a nonzero vector that is perpendicular to both, *except* under what circumstances? (Be as general as possible.)

(e) In the Phong shading model, the specular contribution to the reflected color is $\rho_s \max(0, \vec{n} \cdot \vec{h})^\alpha L_s$. Why do we take the max in the formula? What would go wrong if we didn't?

(f) In Phong specularity, what is the effect (visually) of increasing the α value?

(g) In Phong specularity, why did we not include the object's surface color (C) in the formula?

Problem 2. (20 points) On the extremely distant planet of Omicron Persei 9 (OP9), graphics viewports are designed so that the viewport origin is in the center of the window, with the x-axis directed to the right and the y-axis directed down. Let w and h denote the height and width of an OP9 viewport. (See the figure below right.) Consider a rectangular drawing region whose left and right sides are $x_{\min}$ and $x_{\max}$ and whose bottom and top sides are $y_{\min}$ and $y_{\max}$.



Give the viewport transformation that maps a point $P = (p_x, p_y)$ in the rectangular drawing region to the corresponding point $V = (v_x, v_y)$ in the OP9 viewport. Express your transformation as two equations:

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

Problem 3. (10 points) Consider a hyperboloid in 3-space defined by the following equation:

$$z + \frac{x^2}{a^2} - \frac{y^2}{b^2} = 1$$

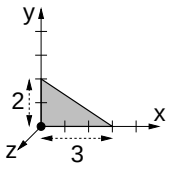
(for some nonzero constants a and b). Given a point $P_0 = (x_0, y_0, z_0)^T$ on the surface of this hyperboloid, derive a normal vector for such a point. You do not need to normalize your vector to unit length. (If the constants a and b confuse you, you may assume that $a = b = 1$ for partial credit.) Show how your derived your answer.

Problem 4. (25 points) Suppose that you are given a function drawWing(), which draws the wing shape shown in the figure below left. (This should be drawn on the $z = 0$ plane. In the figure the z-axis pointing up and out of the page.)

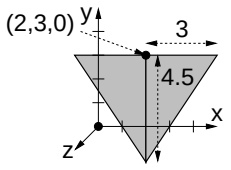
(a) Use the procedure drawWing() and other OpenGL functions (e.g., glPushMatrix(), glRotate*(), glScale*(), glTranslate*(), etc.), to produce a procedure drawBird1() that draws the two wings shown in the center figure.

(b) Explain how to modify your solution to part (a) to produce a procedure drawBird2() that has exactly the same wing shape and size as in part (a), but the two triangles are now rotated up around the bird's central axis, to simulate the flapping of a bird's wings. The angle of rotation is 60 degrees. (Hint: Just show how to modify the solution to (a).)

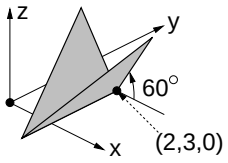
Note: Your procedures drawBird1() and drawBird2() should be performed relative to the OpenGL matrix stack. In particular, their action is transformed by whatever matrix is currently at the top of the matrix stack, and on exit, the contents of the matrix stack should be restored to its original value.



drawWing()
(view from above)



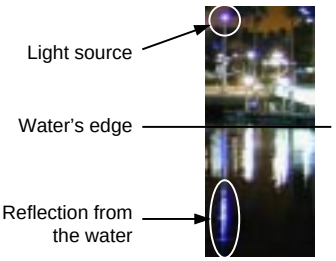
drawBird1()
(view from above)



drawBird2()
(side view)

Problem 5. (15 points)

Consider the image on the right of a picture taken at night of lights being reflected off of water in a harbor. The light source at the top of the image produces a reflection that is very long and thin. Water is a highly specular reflector and poor diffuse reflector.



(a) Based on your knowledge of perspective projection, the Phong lighting model, and the nature of water, explain why the reflected light has this elongated shape. (It may help to draw a picture to illustrate your point.)

(b) Had the water surface been perfectly flat, would the shape of the reflection differ, and if so how?

Problem 6. (0 points) On what television show would you hear about the distant planet of Omicron Persei 9?

Homework 3: Ray Tracing and 3-d Textures

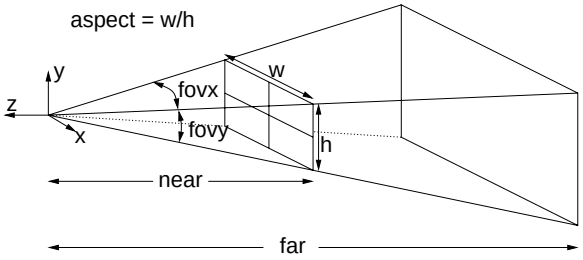
Handed out Tuesday, May 4. Due at the start of class Tuesday, May 11. Late homeworks are not accepted, so turn in whatever you have done.

This is a "no-penalty" homework. Here is how it works. We compute your homework average with and without this homework. If it improves your overall average, then it is included. If not, then it is ignored. These questions are good practice for the final exam, so it is a good idea to attempt them, even if you do not intend to turn it in.

Problem 1. Your boss at Dyno-Graphics Corp. has been informed by marketing that consumers would prefer to specify the x-field of view, rather than the y-field of view. Unfortunately OpenGL does not support this feature. You are given the task of writing a new perspective function, which is given the same arguments as gluPerspective(), but with an x-field of view (in degrees), and you are generate a call to gluPerspective which generates the equivalent y-field of view, based on the other arguments. Here are the function prototypes. Recall that the aspect ratio is the window width over window height.

```
void gluPerspective(double fovy, double aspect, double near, double far)
void dynoPerspective(double fovx, double aspect, double near, double far)
```

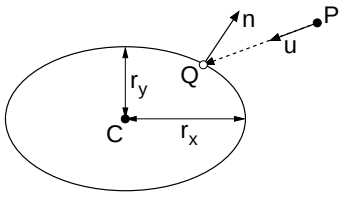
(Hint: This is not as simple as saying fovy $=$ fovx$/$aspect).



Problem 2. Fog is a relatively easy enhancement to a ray tracer. Fog is defined by three parameters, fogStart, fogEnd, and the fog RGB color F. Let C be the color returned by the ray tracing procedure (ignoring fog). Let d be the distance from the ray origin to the point of contact. If d is less than fogStart then C is used, if d is greater than fogEnd then F is returned. Otherwise, an appropriate mixture of the two colors is returned. Give pseudocode for a function, which returns the fog color, given the following parameters: the ray origin P, the ray contact point Q, the traced color C, and the other fog parameters fogStart, fogEnd, and F. You may use any of the utility functions provided in the Color.h file.

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

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

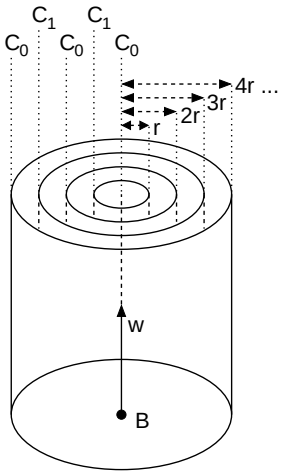


(a) Reduce the ray intersection problem to a quadratic equation, and derive the values of the two roots

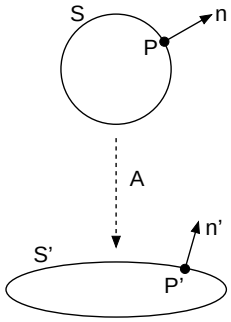
(b) Explain how to determine which root leads to the first intersection point with the ray, and whether the ray hits from the inside or the outside, or misses.

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

Problem 4. The objective of this problem is to develop a 3-dimensional texture for a cylindrical gradient texture. The cylindrical gradient is defined by two colors C_0 and C_1. The geometrical structure is defined a base point B, a central vector $\vec{w}$, which you may assume is a unit length vector, and a radius r. (See the figure below. Note that, unlike the figure, the axis need not be parallel to the coordinate axis.) Consider the series of concentric cylinders of radii 0, r, $2r$, $3r$, ..., growing out from around the axis line passing through B in the direction $\vec{w}$. At even multiples of r the color is C_0, and at odd multiples it is C_1. In between, the color should vary smoothly from C_0 to C_1 and back again. Given a point Q in the 3-space, write a function that maps Q to the appropriate color. (Hint: First map the point Q to a 1-dimensional quantity based on its distance from the cylinder axis, and then follow the approach used for the 1-dimensional gradient given in class.)



Problem 4



Challenge Problem

Challenge Problem. One alternative to defining arbitrarily oriented objects in ray tracing (as we did in Programming Assignment 4) is to instead define very simple shapes, and apply an affine transformation to scale and rotate them into the desired position. One difficulty with this approach is that normal vectors are not generally preserved under affine transformations. This problem investigates this issue.

Suppose that you have a solid geometric object S in 3-space (e.g., let S be a sphere or a convex polyhedron). Let P be a point on the surface of S. Let $\vec{n}$ be the surface normal vector at P. Let A be any affine transformation in 3-space, and let M be the 4×4 homogeneous matrix that represents A. (You may assume that A is nonsingular, meaning that M can be inverted.)

(a) Let $S' = A(S)$, $P' = A(P)$, and $\vec{n}' = A(\vec{n})$. Show that $\vec{n}'$ is not necessarily normal to the surface S' at point P'? (Give an example. An example in 2-space is sufficient.)

(b) Among the basic transformations (translation, rotations, uniform scaling, nonuniform scaling, and shearing), which preserve normals and which do not?

(c) Suppose that A does not preserve normals. As a function of M, how could you transform any normal vector $\vec{n}$ to a vector $\vec{n}'$ that is guaranteed normal to the surface S' at point P'? (Hint: The solution will be of the form $\vec{n}' = M'\vec{n}$, where M' is some function of M.)

Practice Problems for the Final Exam

The final will be on Mon, May 17, 10:30am-12:30pm. The exam will be closed-books, closed-notes, but you will be allowed two sheets of notes, front and back to use for the exam. These problems have been assembled from old exams and homeworks. They do not necessarily reflect the actual difficulty of problems on the exam or the total length of the exam. (Also, be sure to review material from before the midterm as well.)

Problem 1. Short answer questions.

(a) Suppose that a triangle is clipped to a rectangular window. After being clipped against the window, what is the maximum number of sides that the resulting clipped polygon might have? Draw an example to illustrate the worst case.

(b) What is the difference between a left-handed and a right-handed 3-dimensional coordinate system?

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

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

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

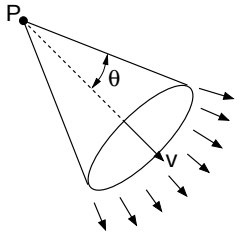
(f) You want to know whether a point P lies on a given surface. From which representation of the surface is this question easier to answer: *implicit* or *parametric*?

(g) Let P be a point on a Bézier curve of degree 3. True or false: The curve has C^4 parametric continuity at this point.

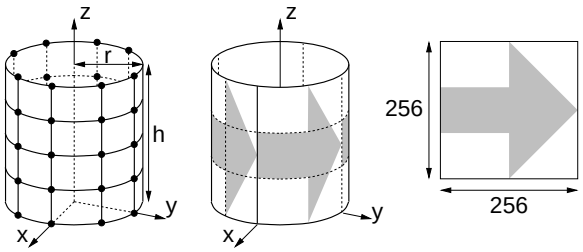
(h) Fill in the blank: "The complex point $c = a + bi$ lies in the Mandelbrot set if and only if the Julia set generated by c is _____."

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

Problem 2. Consider a new type of light called a *spot-light*. A spot-light is defined by giving a point P, a vector $\vec{v}$ (normalized to unit length), and an angle θ. The spot light illuminates any point that lies within an infinite 3-dimensional cone whose apex is P and whose angular radius about $\vec{v}$ is θ. Write a function which, given a point Q in 3-space, and P, $\vec{v}$, and θ, determines whether Q is illuminated by the spot-light.



Problem 3. Your boss at Acme Graphics Corp. wants you to write a procedure to generate a rendering of a cylinder in OpenGL. The cylinder is centered along the z-axis, has a height of h units, and has a radius of r units. Because OpenGL can only display polygons, you are to split the cylinder into v_s vertical stacks (along the z-axis) and r_s radial slices (around the z-axis). (For example, in the figure below left, $v_s = 4$ and $r_s = 8$.) Draw each face as a GL_POLYGON.



(a) Give a procedure (in pseudocode):

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

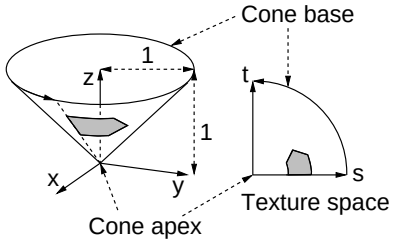
to draw such a cylinder in OpenGL. (You may NOT use any GLUT procedures.) For full credit, you should specify both the vertices and associated normals, so that the shading of the cylinder will be smooth. You do not need to draw the top and bottom of the cylinder.

(b) Your boss also wants you to wrap a texture around your cylinder. The texture image is 256×256 pixels. The texture should be mapped using GL_REPEAT, so that exactly four copies of the image go all the way around the cylinder. Explain how to modify the above procedure to provide the proper texture coordinates for each vertex. (You do not need to give any of the other OpenGL texture commands.)

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. Suppose that you are given two spheres S_1 and S_2, with respective center points C_1 and C_2, both with the same radius r. Assume that the distance between C_1 and C_2 is less than $2r$. Define object $X(C_1, C_2, r)$ to be the "lens-shaped" intersection of these two spheres. Consider a ray $P + t\vec{u}$. Write a procedure that, as a function of C_1, C_2, r, P and $\vec{u}$, computes the parameter value t of the first intersection of the ray with object X. You may assume that you already have access to a function that returns the parameter values t_0 and t_1 of the intersection of the ray with a sphere, where $t_0 \leq t_1$.

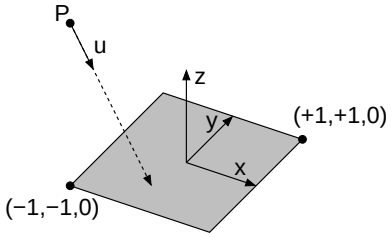
Problem 6. Consider the cone shown in the figure below. Its axis is along the z-axis, its apex is at the origin, its height is 1, and its base has a radius of 1. Also consider a texture that consists of a quarter circle of radius 1. The objective is to wrap this texture around the cone (like wrapping a cover around an ice cream cone) so that the texture origin is mapped to the apex of the cone and the circular arc is mapped to the cone's circular base. The seam where the texture wraps around on itself lies in the xz-plane.



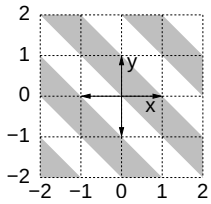
(a) Give an implicit function $f(x, y, z) = 0$ that describes the surface of the cone. (Don't worry about trimming the cone at its base. The infinite surface is good enough.)

(b) Give the *inverse wrapping function*, which maps a point (x, y, z) on the surface of the cone to a corresponding point (s, t) in texture space. (Hint: It may be easier to first derive the polar coordinates in texture space, and then convert to (s, t) coordinates.)

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.

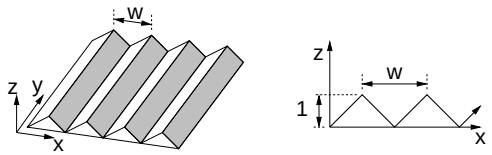


Problem 8. Give pseudocode (or a mathematical formula) for the diagonal strip 2-d texture function shown in the figure below. The dark color is C_0 and the white color is C_1. The horizontal and vertical width of each strip is 1 unit (and hence the diagonal width is $1/\sqrt{2}$).

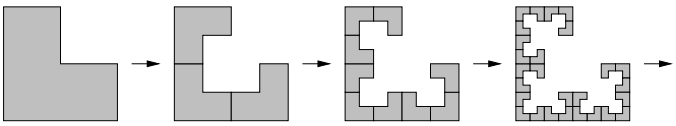


Problem 9. We discussed procedural textures in class. It is also possible to define a *procedural bump map*. (Recall that a bump map does not actually change the shape of a surface. Rather, it generates a perturbed normal vector for each surface point, so that the result of shading appears bumpy.)

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



Problem 10. Consider the sequence of shapes shown in the figure below. What is the fractal dimension of the final (limiting) object? What is its area?



Problem 11. Consider three control points $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$ in the plane.

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

(b) Show that the derivative of this curve, as a function of u, is a Bézier curve of degree 1. In particular, express the derivative as a linear combination of the degree-1 Bézier blending functions.

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; 2–5 points each) Short answer questions. Explanations are not required, but may be given for partial credit

(a) Give a 4×4 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) True or False: For all vectors $\vec{u}$ and $\vec{v}$ in 3-dimensional space, $\vec{u} \times \vec{v} = \vec{v} \times \vec{u}$. If you answered false, what relationship (if any) is there between these two cross products?

(c) Given the homogeneous vector (x, y, z, w), what is the result of applying *perspective normalization* to this vector?

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

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

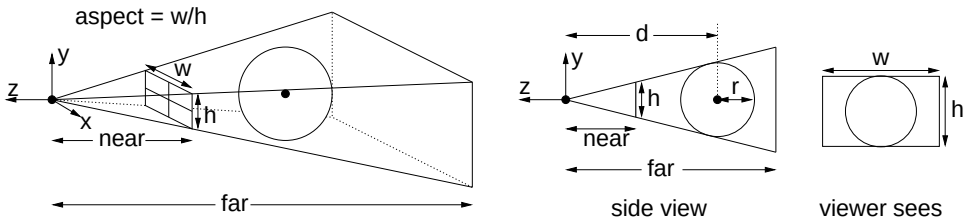
(f) Define the *angle of incidence* between a ray and a surface to be the acute angle between the ray's direction and the surface normal at the point of contact. As a ray goes from a medium of higher index of refraction to one of lower index of refraction does the angle of incidence tend to increase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

(g) Explain the meaning of a *regularized boolean operation*.

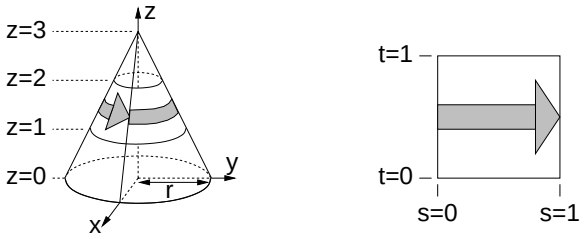
(h) Among the following surfaces, which (if any) is the hardest to apply a 3-dimensional procedural texture: plane, sphere, torus? Briefly explain. You may assume that the contact point and normal vector are given.

Problem 2. (10 points) On the distant planet of Omicron Persei 8, they prefer a different way of specifying the perspective transformation. As with gluPerspective, they give the distances to the near and far clipping planes and the window's aspect ratio, $a = w/h$. However, rather than giving the y-field of view, they instead give the distance d to a sphere of a given radius r that is centered along the viewing direction. The y-field of view is set so that the sphere exactly fills the window's height. (See the figure below.) Write a procedure which, given these parameters, produces an equivalent call to gluPerspective. You may assume that the window is wider than tall, that is, $a \geq 1$. Here are the function prototypes.

void op8Perspective(double d, double r, double aspect, double near, double far)
void gluPerspective(double fovy, double aspect, double near, double far)



Problem 3. (10 points) 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. (15 points) Consider a surface in 3-space defined by the following equation:

$$z + x^2 - 2y = 1.$$

(a) Given a ray $R : P + t\vec{u}$, where $P = (P_x, P_y, P_z)$ and $\vec{u} = (u_x, u_y, u_z)$ derive the t value of the first point of intersection between the surface and the ray. Express your answer by first deriving a quadratic equation of the form $0 = at^2 + bt + c$ and then explain how to compute the roots of this equation in order to determine the first intersection point.

(b) Derive the value of the (normalized) normal vector $\vec{n}$ at this intersection point. There is no notion of outside hit or inside hit, but the normal should be directed to the same side of the surface from which the ray approaches.

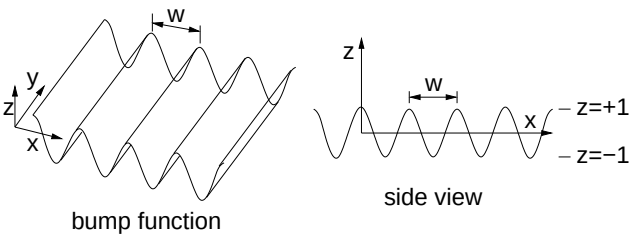
Problem 5. (15 points) This problem considers the derivation of a *procedural bump map*. Recall that a bump map does not actually change the shape of a surface. Rather, it generates a perturbed normal vector for each surface point, so that the result of shading appears bumpy.

Consider the rippled bump function shown in the figure below lying on the x, y-coordinate plane. The ridges run parallel to the y-axis. The height of the ripples alternates smoothly between $+1$ and -1, and at $x = 0$ its height is exactly 1, and the distance between the tops of two consecutive ripples is w. (See the right part of the figure.)

Derive a function, $n(x, y)$, which given the (x, y)-coordinates of a point, returns corresponding the 3-dimensional normal vector $\vec{n}$ for this point. The normal vector should be normalized and directed so it has a positive z-coordinate.

Hint: It may be useful to remember that the following derivative formulas for any a and b:

$$\frac{\partial(a\cos(bx))}{\partial x} = -ab\sin(bx) \qquad \text{and} \qquad \frac{\partial(a\sin(bx))}{\partial x} = ab\cos(bx).$$



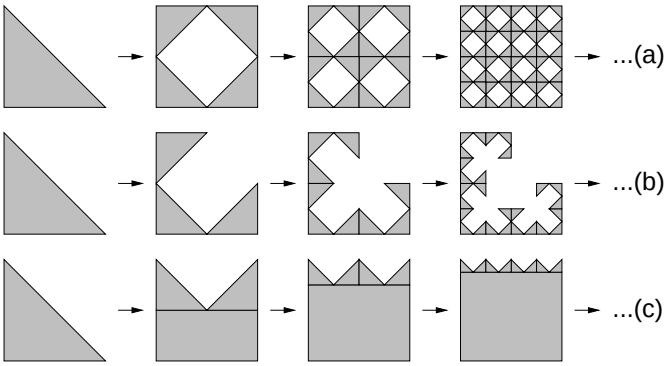
Problem 6. (10 points) Consider three control points $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$ in 3-dimensional space.

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

(b) Compute the derivative of this curve, as a function of u and evaluate this derivative at the point $u = 0$. Express your answer as a function of $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$.

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

Problem 7. (10 points) Consider the three sequences of shapes shown in the figure below. In each case, (a), (b) and (c), derive the fractal dimension of the final (limiting) object. Among the three final shapes, indicate which one(s) are fractals (according to the definition given in class) and which are not. (You may express your answers as a ratio of logarithms.)



Programming Assignment 1: Getting Started

Handed out Tue, Feb 10. The program must be submitted to the grader by Tue, Feb 17 (any time up to midnight). Submission instructions will be coming. 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.

In this assignment you will be given a simple 2-dimensional OpenGL program, and will modify it to listen to a number of simple user inputs. The initial program, called prog1-start.cpp can be downloaded from the class web page. This program creates a 400×400 window. It assumes that the idealized drawing area is a 2×2 square ranging from the lower left corner $(-1, -1)$ to upper right corner $(+1, +1)$. The initial image contains a blue rectangle on top of a red diamond, both centered in the middle of the window. The blue rectangle has a side lengths of 1 and the red diamond has height and width 1.6. Every 10 seconds the colors of these two shapes are swapped. Also, if the left mouse button is clicked, then the colors swap. The window can be resized and covered and uncovered. Whenever it is resized, the shapes are resized in a corresponding manner. (That is, if the window is made tall and skinny, then so are the two shapes.) When the 'q' key is hit the program quits.

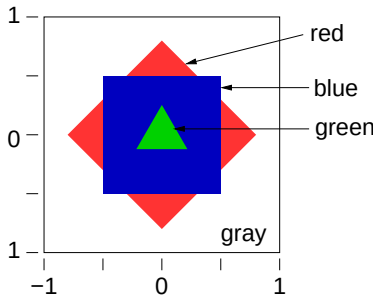
Your task is to modify this program so that it retains its current features, and adds the additional features given below. These are broken into two groups. The basic enhancements are worth roughly 80% of the credit. The additional enhancements are worth the remaining 20%.

Original Features:

- Colors swap every 10 seconds.
- Left-button click swaps the colors.
- Resize and redisplay callbacks are handled.
- 'q' quits the program.

Basic Enhancements:

- Draw a green equilateral triangle of side length 0.4 centered in the middle of the window.



- Pressing the right mouse button changes the triangle color to black and releasing the mouse button changes it back to green.
- Pressing any arrow key moves the rectangle in the corresponding direction by 0.02 units. (Hint: Check out the Glut function glutSpecialFunction() for further information on processing arrow and function keys.)
- CONTROL + any arrow key move the diamond by 0.02 units. (Hint: Check out the Glut function glutGetModifiers() for further information on how to determine whether the Control key is depressed. To test this condition, the function's result is logically "anded" with GLUT_ACTIVE_CTRL bit, and then tested for being nonzero.

Additional Enhancements: Each of these additional enhancements is worth 5% of the grade.

- Hitting the 'r' and 'R' keys cause the rectangle to be rotated by 5 degrees counterclockwise and clockwise about its center, respectively.
- Hitting the 'd' and 'D' keys cause the diamond to be rotated by 5 degrees counterclockwise and clockwise about its center, respectively.
- Holding down the left mouse button and dragging the mouse translates the rectangle. Its center should follow the cursor.
- Holding down the right mouse button and dragging the mouse translates the diamond. Its center should follow the cursor.

I will leave the task of computing the vertex coordinates of the equilateral triangle as a geometric exercise. (Remember from your high school geometry that the center of an equilateral triangle splits the triangle's altitude in the ratio $1/3 : 2/3$.)

There are a couple of ways to move the rectangle and diamond. One is to use the OpenGL matrix transformations discussed in class and the other way is to recompute the vertex coordinates with each redrawing. You are allowed to choose whichever mechanism you prefer.

To perform the dragging operation check out the function glutMotionFunc(). Beware that Glut uses the upper left corner as the window origin and OpenGL uses the idealized coordinates when drawing. It will be up to you to make the necessary coordinate transformations.

You can extra credit points for adding additional enhancements. Recall that extra credit points are not part of the basic grade, and only considered after the final cutoffs have been assigned. The number of credit points are determined subjectively by Pooja, based on the degree of creativity and effort involved. You are welcome to be creative and add whatever features you like. Some ideas include:

- A different behavior for window resizing, which preserves the relative object shapes better.
- Some interesting animation effect involving the shapes.
- Implementing your program in Java, using JOGL and the Java AWT.

Programming Assignment 2: 2-Dimensional Flocking

Handed out Tue, Mar 2. The program must be submitted to the grader by Tue, Mar 16 (any time up to midnight).

Overview. Many video games and physical simulation systems involve planning and coordinating the motion of a groups of objects, such as birds in a flock or a school of fish. The purpose of this assignment is to implement a simple 2-dimensional program that models the behavior of a group of moving objects in the plane, called *boids*.

Flocking motion satisfies some basic elements:

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

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

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

The above properties define the nature of the motion at local level, but does not constrain the groups overall motion. The animator must be able to control the *global motion* of the boids as well. For example, the boids should move towards some goal point or follow a leader whose motion is prespecified.

In this assignment, you are to write a program to simulate the motion of a group of two-dimensional boids, subject to the above general requirements. Basic credit will be based on the set of capabilities that you successfully implement. Extra credit will be based on the TA's subjective judgment of how interesting or complex your motion and rendering is.

Program Requirements. Your program must implement the following basic elements for 70% credit.

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

Global trends: In order to implement global control of motion the boids will attempt to follow a leader boid. This leader boid will be determined by the location of the mouse cursor in the graphics window.

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

For full credit, add each of the following elements for the specified amount of credit.

Remaining in the window: (5 points) Boids should not be allowed to fly outside the window.

Pause/Resume: (10 points) By hitting the 'P' key the simulation can be paused and resumed. The '+' and '-' keys should work when the program is paused or running.

Obstacles: (10 points) Some number of disjoint circular obstacles are present in the scene. The locations of the obstacles are specified in a reasonable way at run time (e.g. by giving their radii and center coordinates in an input file). Boids should avoid obstacles. If the flock is split by flying around a small obstacle, it should regroup in a natural way.

Object Rendering: (5 points) When drawing each boid, do so in a manner that indicates the direction of flight. (E.g., each boid can be rendered as a thin triangle or thin parallelogram that is pointed in the direction of flight.)

Some ideas for extra credit:

Flapping motion: Draw your boids so they look like birds flying, fish swimming, animals walking, or whatever you like. In the best case the motion should be realistic (e.g., all birds do not flap at exactly the same time).

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

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

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

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

$$\vec{v} = \begin{pmatrix} s\cos\theta \\ s\sin\theta \\ 0 \end{pmatrix}$$

and

$$s = \|\vec{v}\| \qquad \text{and} \qquad \theta = \arctan(v_y/v_x).$$

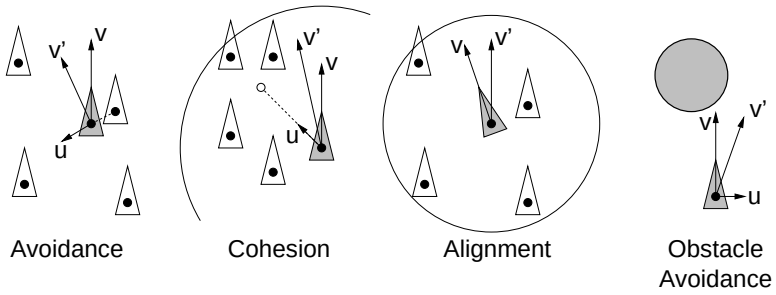
(The best way to compute the arc tangent is using the built-in function atan2(vy, vx), which returns an angle from $-\pi$ to $+\pi$.) Whether you choose to represent velocity using s and θ or using $\vec{v}$ is up to you. Whichever you choose, you will probably need to be able to convert to the other.

In order to do the animation, you will need to set up a continuous event loop. This can either be done using glutTimerFunc() (with a small time delay, say 1/30 of a second) or glutIdleFunc(). Each time this callback is invoked, your program will update the locations of the boids, by adding the current velocity vector to their current location and calling glutPostRedisplay().

Rather than adjusting the location of the boid at each step, it is better to adjust the boid's velocity vector incrementally, and then move the boid according to this velocity vector. This results in a smoother motion. Each step of the animation involves the following elements:

(a) For each boid, determine the other boids and obstacles that close by (within some fixed distance, and perhaps giving higher priority to things that are in front).
(b) Update the velocity vector for each boid in order to satisfy the various flocking and obstacle/avoidance constraints.
(c) Move each boid by adding an appropriate scaled copy of its velocity vector.
(d) Redraw the scene.

The most challenging step is (b). Each of the various flocking properties has a certain "pull" on the velocity vector. For example, if you are too close to another boid, this tends to push the velocity vector away from this other boid. (See the figure below.) For cohesion, compute the locations of the boids within some limited radius and compute the centroid (center of mass) of these locations. The boid's velocity vector will be set to move it towards this centroid. For alignment, compute the average of the velocity vectors of the boids in some limited distance, and change your velocity vector so that it is closer to the average. To avoid obstacles, check whether there is an nearby obstacle in front of the boid. If so, you want to turn the boid in the direction that most easily avoids the obstacle. This turn has the effect of influencing the current velocity vector.



In summary, there are a number of "corrections" to the current velocity. In order to determine the updated velocity, assign weights to these corrections and then add the weighted sum to the current velocity vector. The assignment of weights is a nontrivial task. Some forces (e.g., the desire to avoid obstacles or predators) are much stronger than others (e.g., the desire to maintain cohesion). It is probably a good idea to put some upper bound on how much the velocity can change, to avoid unreasonably fast velocity changes.

Further information on Boids and flocking behavior can be found on the web. I will post a link to a tutorial from a course given at SIGGRAPH conference on the class web page. There are also demo programs and animations on boids and flocking behavior to be found on the Web. These are useful for getting ideas, you are required to do your own implementation for this programming assignment.

Programming Assignment 3: 3-Dimensional Flocking

Handed out Thu, Apr 1 (updated Wed, Apr 7). The program must be submitted to the grader by Tue, Apr 20 (any time up to midnight).

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

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

Steerable Goal: Rather than having the goal simply follow the mouse, the goal point will be a flying point. Through keyboard inputs (or mouse, if you prefer) the user controls the flight path of the goal.

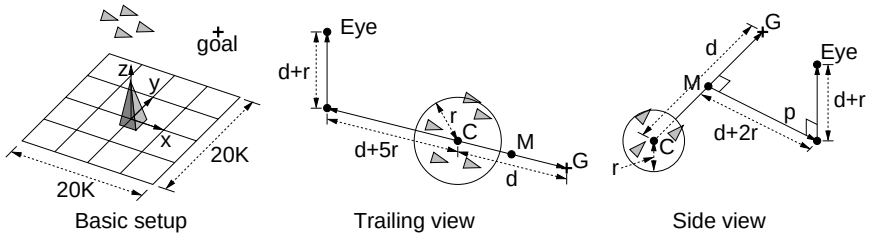
Multiple Views: The camera can be positioned in various locations, one at a fixed location, one behind the boids, and one to the side.

3-dimensional Obstacles: Obstacles will consist of spheres and cones. Boids must avoid obstacles by flying around them.

Graphics effects: The scene will be lighted. The program will also have options for activating other visual effects, such as texture mapping, fog, and ground shadows.

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

General setup: The ground lies along the x, y coordinate plane and the z axis points up to the sky. There should be a large ground terrain, which will provide your boids ample area to fly over. There is no requirement that the boids stay within this region, however. Near the center of the domain there should be an observation tower. (See the figure below, left.)



For example, our ground was a large square on the x, y plane of side length $20,000$ units. It is rendered as an alternating 50×50 checkerboard pattern of squares, and thus each square is 400 units on a side. Our boids have a speed of around 40 units per second and are roughly 10 units in length.

Create an initial flock of boids. These can be placed randomly in a confined region of space. (We started ours with 10 boids placed randomly in a roughly $50 \times 50 \times 50$ box near the point $(2400, 150, 1200)$.)

The initial goal should be placed at a moderate distance away from the initial flock and should be given an initial velocity, so that the simulation starts in a reasonable state. (Ours was at $(2000, 0, 1000)$ with velocity $(-40, 0, 0)$.)

Steerable Goal: The goal point is no longer tied directly to the mouse. Instead it flies through space under the user's control. If you do nothing, the goal flies at a constant velocity. You can slow it down and speed it up and turn it left and right, up and down through some combination of keyboard and mouse input. (We used 'v' and 'b', for slowing down and speeding up and used the arrow keys for steering always, and in trailing view mode we used mouse input as well.) For testing purposes, it should be possible to fly your goal anywhere: through obstacles, under the ground, outside the scene.

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

Implementation note: There are a few ways to handle the goal control. We implemented our goal much like a boid, with a position P_g and a velocity vector $\vec{v}_g$, but the goal does not observe any of the boid behavior rules. Incremental speeding up and slowing down were handled by scaling $\vec{v}_g$ vector times an appropriate factor. To move the goal velocity up and down, we added (to $\vec{v}_g$) a vector that is parallel to the z axis, and whose length depends on the length of $\vec{v}_g$. To turn left and right, we did something similar, but rather than using the z unit vector, we used a vector $\vec{p}$ of unit length that is perpendicular to both $\vec{v}_g$ and the z unit vector. (This perpendicular vector can be computed using the cross product.)

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

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

Default View: This view is taken from an observer (a bird watcher?) located at a fixed position. (In our case this was from the top of an observation tower, centered over the origin.) The view should be centered about M. You may adjust the field-of-view to simulate a telephoto lens, but this is not required.

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

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

Side View: This view is taken from the right side of the vector from C to G and above.

For example, our approach was to consider a vector $\vec{p}$ that is simultaneously perpendicular to $\vec{u}$ and to the z axis. (This can be computed using the cross product.) Starting from M we move in the direction of $\vec{p}$ by a distance of $d + 2r$. We then move up (parallel to the z axis) by a distance of $d + r$. We took a view centered at M whose x field of view is 40 degrees.

Since these views are always perfect, we found it useful to have controls that would zoom in/out and up/down through keyboard input by modifying the distances used above.

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

Additional Requirements. The following requirements should be added for full credit. The point values are indicated with each one.

Obstacles: (10% total) In addition to the ground, there are two types of obstacles, spheres and cones, around which your boids should fly. The exact locations of the obstacles will be provided by a file, which we will provide. (See the file ReadMe.txt for explanation of the input file format.) You can use GLUT procedures for drawing these shapes. (2% for smooth ground avoidance, 4% for cones and 4% for spheres.)

3-dimensional Boids: (5%) Boids should be drawn as 3-dimensional polyhedra. As before, they should face the direction in which they are flying.

Shadows: (3%) Draw a shadow of each boid on the ground. (You are not required to draw the shadows that boids cast on each other or other objects.) This can simply be a vertical projection of the boid on the ground, but more realistic shadows will be given extra credit.

Implementation Note: The shadows should be drawn slightly above the ground, because of errors in depth computations.

Flapping: (5%) Animate your boids so they flap their wings. For the sake of realism, boids should not all flap at the same time.

Texture mapping: (5%) Have an option (e.g. through keyboard input) that activates and deactivates some form of texture mapping. Supply whatever texture image files you use as part of your submission. Because texture mapping can slow down your program considerably (on our weak test machines), be sure that it is possible disable this feature.

Fog: (3%) Have an option (e.g. through the keyboard input) that activates some sort of fog effect.

Pause/Single Step: (5%) The user should be able to pause the program, advance it by single step (e.g. by hitting the space bar), and then resume it to continuous mode. This is important for debugging and our testing.

Reshape: (4%) The user should be able to resize the window. When the window is redrawn the image should not be distorted. (That is, scaling must be performed uniformly.)

Ideas for Extra Credit. Here are some ideas for extra credit. Feel free to be creative and invent those of your own. Beware to avoid enhancements that involve OpenGL extensions, since we will likely not be able to test them.

More interesting scene: Create a more interesting environment. This can be done by altering the shape of the scene and/or by creating texture maps to simulate distant object. It is not required that you implement obstacle avoidance for all the objects in your scene.

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

Better shadows: Have the shadows depend on the location of the light sources, or cast shadows onto non-ground objects.

Game Mode: If you are using a PC, have an option to run in full screen mode. (See glutEnterGameMode(). Information on this can be found on the web. (Be sure that this option can be disabled, because it is rather flaky from one machine to another.)

Split Views: Rather than switching between views, have an option that creates multiple viewports showing the various views.

Perching: When the boids come close to the ground they should slow down, land momentarily, and then after a little time fly off towards the goal again. (See `http://www.vergenet.net/~conrad/boids/pseudocode.html` for further information.)

Programming Project 4: Simple Ray Tracer

Handed out Tue, Apr 20. The program must be submitted to the grader by Tue, May 11 (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, cylinders and cones, checkerboard texture, and reflection and refraction. For extra credit you may add other features, including other object types, other types of textures, texture mapping, and antialiasing.

This program does not involve OpenGL or Glut. (It can be implemented equally easily in C++ or Java, but note that there is a lot of affine geometry involved, and you may find our Geom3d package useful, which is written in C++.)

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:

```
Prog4 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.

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.

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 $n_\ell - 1$. 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.

```
2                                   # 2 lights
0   0  0   0.5 0.5 0.5   0  0   0   # white ambient light
0  10 50   1.5 0.0 0.0   0  0.1 0   # red light at (0,10,50)
```

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. (We will explain this in class.)

```
2                             # 2 pigments
solid    0.0  0.4  0.0        # Pigment 0: solid dark green
checker  1 0 0    0 0 1  2.0  # Pigment 1: red-blue checker, size 2
```

There is a default color, which we defined to be gray (RGB $= (0.5, 0.5, 0.5)$). If no object is hit, then the default 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, $\langle \rho_a, \rho_d, \rho_s, \alpha, \rho_r, \rho_t, \eta_t \rangle$. These are the ambient coefficient ρ_a, the diffuse coefficient ρ_d, the specular coefficient ρ_s, the shininess α, the reflectivity coefficient ρ_r, and transmission coefficient ρ_t, and finally the index of refraction of the object's interior η_t. 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 ρ_r, ρ_t and η_t, but you still have to input them. (See Lecture 18 for more information.)

```
2                                   # 2 surface finishes
0.3  0.1  1.0 500  0.9  0.0  0      # 0: highly specular and reflective
0.0  0.7  0.0  50  0.0  0.5  1.5    # 1: diffuse, partially transparent
```

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:

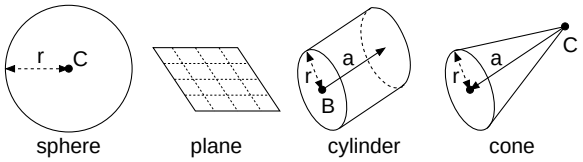
Sphere: The word "sphere" followed by the (x, y, z) center coordinates radius r.

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 $\langle 0, 0, 1, 2 \rangle$.

Cylinder: The word "cylinder" followed by the (x, y, z) of a base point B, the (x, y, z) coordinates of the axis vector $\vec{a}$, and the radius r. (The opposite base point is $B + \vec{a}$.)

Cone: The word "cone" followed by the (x, y, z) of the apex point C, the (x, y, z) coordinates of the axis vector $\vec{a}$, and the radius r at the base. The center of the base of the cone is located at $C = \vec{a}$.

```
2                          # 2 objects
0 0 sphere 2  0 -5   2     # solid sphere at (2,0,-5) with radius 2
1 1 plane  2  1  0  10     # checkered plane 2x + y + 10 = 0
```



You may assume that there are at most 20 light sources, 50 pigments, 50 surface finishes, and 200 objects in a scene.

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

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 contribution only if the ray hits this source before any other object.

Checkerboards: (5%) Implement checkerboard pigments.

Shapes: Include plane objects (5%), infinite cylinders (5%), and infinite cones (5%). For an additional (5%) trim the cylinders and cones to have only a finite extent. (Hint: This can be done by intersecting the infinite shape with two planes.)

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 *convex polyhedra* as the intersection of a number of planes. (The process is similar to the Liang-Barsky line clipping algorithm. For each ray maintain the minimum and maximum t values.) *Triangles* and *circular disks* are also pretty easy to implement. 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 (xv for example, by hitting the middle mouse button). 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, I will be making available code for producing and outputting .bmp files, based on the RGBpixmap object given in our book. This will come in the form of an enhancement to two files, RGBpixmapV2.h, and RGBpixmapV2.cpp.

The first 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.

```
#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
```

Programming Assignment Submission

Your submission will consist of an encapsulation of files, which will be emailed directly to our TA, Pooja Nath, pooja@cs.umd.edu. Please read the following instructions carefully, since significant deviations can result in the loss of points. The encapsulation must include the following items:

Readme.txt: There must be a file called "Readme.txt", which contains information to the grader on how to compile and run your program. This includes:

Your name and email: (Very important.)

System/Compilation Information: What system should the program be run on (e.g. WAM, Linux Lab, PC Visual Studio V.6 or Visual Studio.NET). Also explain how to compile your program (very important). Ideally the grader should just have to enter "make" for Unix/Linux, or invoke the "Build" menu item in Visual C++.

How to run it: Ideally your program should be self explaining. But if it is not, please explain the inputs and how to control the various elements of the program.

Special Features: List special features or extensions, which you would like the grader to consider.

Known Bugs and Limitations: List any known bugs, deficiencies, or limitations with respect to the project specifications. As a service to the grader, please be complete here. If she finds bugs that were not listed here, she reserves the right to double the normal deduction.

File directory: If you have multiple source or data files, other than those created by the compiler, please explain the purpose of each file.

Source Files: All the source files and header files. Just what we need for compilation. (Do *not* include files that we already have, such as glut.h or jogl.jar.)

Makefile or .dsw/.dsp: On Unix you should have a Makefile, which compiles your program. For Visual Studio, you should include the workspace/project/solution files, whatever is needed for compilation. A sample can be found on the class OpenGL web page.

Input Files (if applicable): If the program requires input data, provide some test data files.

DO NOT INCLUDE: Because the grader's disk quota is limited, please delete all executable and object (.o) files prior to submission. For Visual C++, check that your Debug and Release directories are empty. You can clean these by selecting "Build→Clean".

Important Note: Irrespective of which platform you developed your system on, it must be executable from a WAM, LinuxLab, or PC in the WAM lab (using Visual C++). Do not assume that all systems are compatible. I would urge you to download your latest version of the program from your development system to one of these systems every day or so, just to test for compatibility. If your program produces compilation or execution errors, you will asked to resubmit, and suffer the resulting late penalties.

How to submit: First, store everything (Readme.txt, Makefile, source) in a directory whose name easily identifies you e.g. "JoeSmith". In particular, *avoid* names like "prog1", since it will make it harder for the grader to keep track of submissions. Be sure to delete any unnecessary files (executable or ".o" files).

Then email everything to the TA. There are a couple of ways to do this.

Use the submit427 script: The following script is available on WAM Unix or LinuxLab machines:

```
~mount/submit427 JoeSmith
```

where "JoeSmith" is the name of the directory with your files. The script prints out messages as it executes. Watch for any error messages as the script runs. (It is not very robust.)

Zip'd attachment: This is just a manual version of the previous.

(1) Encapsulate everything into one file, for example by using tar and gzip:

```
tar -cvf JoeSmith.tar JoeSmith
gzip JoeSmith.tar
```

or zip:

```
zip -r JoeSmith JoeSmith
```

or any standard PC bundling software, like WinZip.

(2) Send the resulting bundle as an email attachment to the grader (pooja@cs.umd.edu).

If you discover an error in an earlier submission, you may repeat your submission. But in consideration to the grader's time and disk quota, please keep the number of submissions to a minimum. She will grade the last submission she receives. Server errors are rare but can occur. Be sure to save your final submission somewhere safe (very important).

Check the Next Day: Please check your email the next day after submitting. If the grader has problems unpacking your files or compiling your program, she will ask you to resubmit. The grader reserves the right to deduct points if you did not follow the above procedures or if your program does not compile on one of the approved platforms.

Late Submissions: Programs are due by midnight of the due date. Late programs will be subject to the deductions given on the syllabus (up to six hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for every additional day late.) The evenings of major project submissions are typically times of heavy load on the systems. You are strongly urged to get the program running well in advance of the deadline, and save the last days for cleaning up the code and writing documentation.

Documentation: As in all programming courses, you will be graded in part for clean structure and good program documentation. Clean structure means designing a program that is elegant, efficient, and easily understood to someone reading the source. Good documentation should provide a clear explanation for someone who wants to understand what your project does and how it works. Some things to include are:

File Header: At the start of each file, there should be short comment that says what the file contains, who wrote it, and what purpose it was written for.

General Structure: As an aid to the grader, there should be at least one large comment that provides a general overview of your program, its major structures, and how it is organized.

Section Header: Each major procedure or segment of code (e.g. roughly once every page) should have a comment explaining what this segment of code does. Major procedures should have information describing argument lists and whether any global data is modified.

In-line comments: Every few lines of code, it is a good idea (unless the code is completely transparent) to explain what is going on.

Using OpenGL on Local Machines

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.

WAM Unix: The Sun Solaris workstations in the WAM labs located throughout campus. Virtually all of this information applies to the machines in the Glue labs as well.

CSIC Linux Lab: The Dell machines running Redhat Linux in the CSIC Linux Lab on the third floor of the CSIC building.

WAM PC/Windows: The PC's running Microsoft Windows in the A.V. Williams WAM lab.

PC Windows: Your own PC running Microsoft Windows.

To understand this more concretely, while you are reading this you should also download the sample program, which we have made available. From the class web page go to the "OpenGL" link at the top of the page, and then follow the link to the "Sample OpenGL Program," or go directly to the following link.

http://www.cs.umd.edu/~mount/427/OpenGL/OpenGLSample

More detailed information can be found in the "Readme" files contained within the bundle.

OpenGL is the most widely used graphics library standard, that is, it is just a specification for a graphics library, which has been implemented by a number of vendors. OpenGL consists of two principal components: GL (basic OpenGL) and GLU (OpenGL utilities). GL is responsible for the basic low-level rendering tasks, and GLU provides support for some higher-level operations, such as drawing curved surfaces. (There are also other related components, GLX or WGL, which are used for handling special extensions to basic OpenGL.)

In addition, it is necessary to use a toolkit for creating windows and handling user interaction. For C/C++ programming, we will use GLUT (OpenGL utility toolkit). Java programmers will need to use a different toolkit, for example, the Java AWT (Abstract Window Toolkit).

Sun Workstations in the WAM/Glue Labs: OpenGL and Glut are installed on the WAM and Glue. (The version of OpenGL is Mesa, an open-source implementation of OpenGL.) These machines are easy to use, very reliable, and convenient, but the graphics is not the best, and complex 3-d graphics will tend to run slowly. Nonetheless, these can be used for your initial development. (In fact, it is often easier to debug graphics on slow machines, because when something goes wrong, it happens slowly enough that you can see it clearly!)

The library files libMesaGL, libMesaGLU, and libglut are located in /usr/local/Mesa/lib and the include files gl.h, glu.h, and glut.h are located in /usr/local/Mesa/include/GL.

To compile a C/C++ program on these, copy the file Makefile-WAM-Sun to Makefile, and enter "make". Once compiled, you should be able to execute it by entering "sample1". You will need to place the cursor over the window for your keyboard input to be processed by the program.

Remote Execution: If you have an X-server on your PC at home (e.g. XFree86 or Reflection) you can remotely log into the WAM or Glue labs, compile your program, and run it. The graphics should appear on your PC display. Hint: before trying this with an untested OpenGL program, try a known X11 application (for example, enter "xv"). If that works, then try running your program. If everything is configured properly, the graphics should appear on your screen. Beware, it will be very, very slow. But it is an option for your initial development.

CSIC Linux Lab: For machines in the Linux lab, the procedure is essentially the same as above. The only significant difference is where the files are stored. 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" and "whereis" can often be used to help you locate where these files are on any particular Unix/Linux system.

In the CSIC Linux Labs, the library files libGL and libGLU are located in /usr/X11R6/lib and libglut is located in /user/local/freeglut. The include files gl.h and glu.h are located in /usr/include/GL and glut.h is located /usr/local/freeglut/include/GL. (Redhat dropped support for Glut because it was too hard to configure and compile. Freeglut is essentially the same, and much easier to work with.) Also, use Makefile-CSIC-Linux as your Makefile.

PC's in the WAM Lab (A.V. Williams): There are three PC's in the AVW Wam lab that have OpenGL and Glut installed along with Microsoft Visual Studio 6 and Visual Studio.NET. (Warning: .NET is quite slow to launch, and you may want to use Visual Studio just to keep from going crazy.) Instructions have been provided for creating OpenGL programs under both systems. These can be found in the Readme files in the directories VisualCPP (for Visual Studio 6) and VisualStudioNET (for Visual Studio.NET). Alternatively, after you download and unbundle the sample program, double-click on the file VisualCPP/sample1.dsw (for Visual Studio 6) or the file VisualStudioNET/Sample1/Sample1.sln (for Visual Studio.NET).

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

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

$\langle VCpp \rangle$: Your visual C++ directory. For Visual Studio 6 this is something like:

C:\Program Files\MicroSoft Visual Studio\VC98

For Visual Studio.NET this might be:

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

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

OpenGL should already be installed on your machine. To verify that OpenGL is installed on your system, first do a search for the files opengl32.dll and glu32.dll. They should appear in your windows system directory (with lots of other dll files). You need to install Glut, however. The easiest way to do this is to visit the following web page. It contains precompiled glut libraries. (Download the "image datafiles" not the "source code".)

http://www.xmission.com/~nate/glut.html

After unbundling the file, copy the following files to the following directories:

glut32.dll $\Longrightarrow$ $\langle WinDir \rangle$\SYSTEM32 (or wherever opengl32.dll is)
glut32.lib $\Longrightarrow$ $\langle VCpp \rangle$\lib
glut.h $\Longrightarrow$ $\langle VCpp \rangle$\include\GL.

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

Now, you should be ready to go. If you have Visual Studio 6, then the quickest way to proceed is to go to the directory VisualCPP and double click the workspace file sample1.dsw. If you have Visual Studio.NET, go to the directory VisualStudioNET and double click the solution file Sample1/sample1.sln.

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

Lighting in OpenGL

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

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

glEnable(GLenum cap), glDisable(GLenum cap)**:**

Enable/disable some option. The following options are useful for 3-dimensional hidden surface removal and lighting. By default, all are initially disabled.

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

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

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

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

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

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

Lighting: In OpenGL there may be up to 8 (officially GL_MAX_LIGHTS) light sources (GL_LIGHT0 through GL_LIGHT7). If lighting is enabled (see glEnable()) then the shading of each object depends on which light sources are turned on (enabled) and the materials and surface normals of each of the objects in the scene. Note that when lighting is enabled, it is important that each vertex be associated with a proper normal vector (by calling glNormal*()) prior to generating the vertex.

glShadeModel(GLenum mode)**:**

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

glLightModelf(GLenum pname, GLfloat param)**:**
glLightModelfv(GLenum pname, const GLfloat *params)**:**

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

glLightf(GLenum light, GLenum pname, GLfloat param)**:**
glLightfv(GLenum light, GLenum pname, const GLfloat *params)**:**

Defines parameters for a single light source. The first version is for defining scalar parameters, and the second is for vector parameters. The first argument indicates which light source this applies to. The argument pname gives one of the properties to be assigned. These include the following:

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

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

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

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

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

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

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

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

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

GL_EMISSION	(vector) RGBA of the emitted coefficients
GL_AMBIENT	(vector) RGBA of the ambient coefficients
GL_DIFFUSE	(vector) RGBA of the diffuse coefficients
GL_SPECULAR	(vector) RGBA of the specular coefficients
GL_SHININESS	(scalar) single number in the range $[0, 128]$ that indicates degree of shininess.

Shade Model: Because OpenGL only deals with flat objects, programmers need to use many small flat polygonal faces to approximate smooth surfaces, such as spheres, say. But this raises the question of whether the user wants the object to appear smoothly shaded or to clearly see the boundaries between adjoining faces. This is done through the shading model, whose argument is either GL_SMOOTH (the default) or GL_FLAT.

```
glShadeModel(GL_SMOOTH);
```

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

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

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

Textures, Fog and Color Blending in OpenGL

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

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

glEnable(GLenum cap), glDisable(GLenum cap)**:**

Enable/disable some option. The following options are useful for texture mapping and fog. More details on controlling these effects are given below. By default, all are initially disabled.

GL_FOG**:** Enables fog.

GL_BLEND**:** Enables color blending (using the 'A' in RGBA) to achieve transparency and related effects.

GL_TEXTURE_2D**:** Enables texture mapping.

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

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

glBlendFunc(GLenum sfactor, GLenum dfactor)**:**

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

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

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

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

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

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

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

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

glTexImage2D**:** (GLenum target, int level, int internalFormat, int width, int height, int border, GLenum format, GLenum type, void *pixels):

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

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

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

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

There are subtle differences between GL_DECAL and GL_REPLACE when different formats are used or when the 'A' component of the RGBA color is not 1. See the reference manual for details. The default is GL_MODULATE, which is a good choice for combining textures with light.

glTexParameterf(GLenum target, GLenum pname, GLfloat param)**:**

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

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

GL_TEXTURE_MAG_FILTER	magnification filter
GL_TEXTURE_MIN_FILTER	minification filter

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

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

This procedure may also be invoked to specify other properties of texture mapping.

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

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

```
static GLuint texName[5];       // texture names for 5 textures
glGenTextures(5, texName);      // create 5 texture names
                                // make texture 0 the current texture
glBindTexture(GL_TEXTURE_2D, texName[0]);
// ... operations/drawings involving texture 0

                                // make texture 2 the current texture
glBindTexture(GL_TEXTURE_2D, texName[2]);
// ... operations/drawing involving texture 2
```

Texture Mapping Utility: In order to use texture mapping, you must present a texture to OpenGL as an array. Typically, textures are given as image files in some standard format (.jpg, .gif, .ppm, .bmp). There are many programs that can convert from one to another (on the Linux cluster you can use gimp, for example). To help you with the task of inputting images, I have adapted a utility program, which I found in Hill's Graphics book. It consists of a class RGBpixmap that stores an image. Its main method reads in an image from a .bmp file:

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

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

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

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

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

```
...
glEnable(GL_TEXTURE_2D);                        // enable texture mapping
glMaterialfv(GL_FRONT_AND_BACK,                 // white base color
           GL_AMBIENT_AND_DIFFUSE,
           glfv(white));
glBegin(GL_POLYGON);                            // draw the object
    glNormal3f  (/*...specify normal  coordinates for vertex 0...*/);
    glTexCoord2f(/*...specify texture coordinates for vertex 0...*/);
    glVertex3f  (/*...specify vertex  coordinates for vertex 0...*/);
    // ... (repeat for other vertices)
glEnd();
glDisable(GL_TEXTURE_2D);                       // disable texture mapping
...
```

Figure 2: Displaying a texture-mapped object.