6.837 --- FALL '98

Final Quiz

Time allotted: 3 hours

Open book, open notes
(100 points)

Instructions: Answer all questions using the space provided. If you need more paper, raise your hand, and it will be provided for you. If you have a question during the test, do not leave your seat. Instead, raise your hand until someone comes to assist you. If any question seems unclear or appears to lack sufficient information, state a reasonable assumption (in writing on your quiz) and proceed. The point value of each problem is indicated in parentheses. Make sure that you answer each part of each question.

Solutions

Name_____
(Print)

I understand that this quiz is subject to the
conditions and procedures stated in the
MIT Polices and Procedures handbook in Section 10.2.
Furthermore, I neither gave, nor received, any
unsanctioned assistance during this examination.

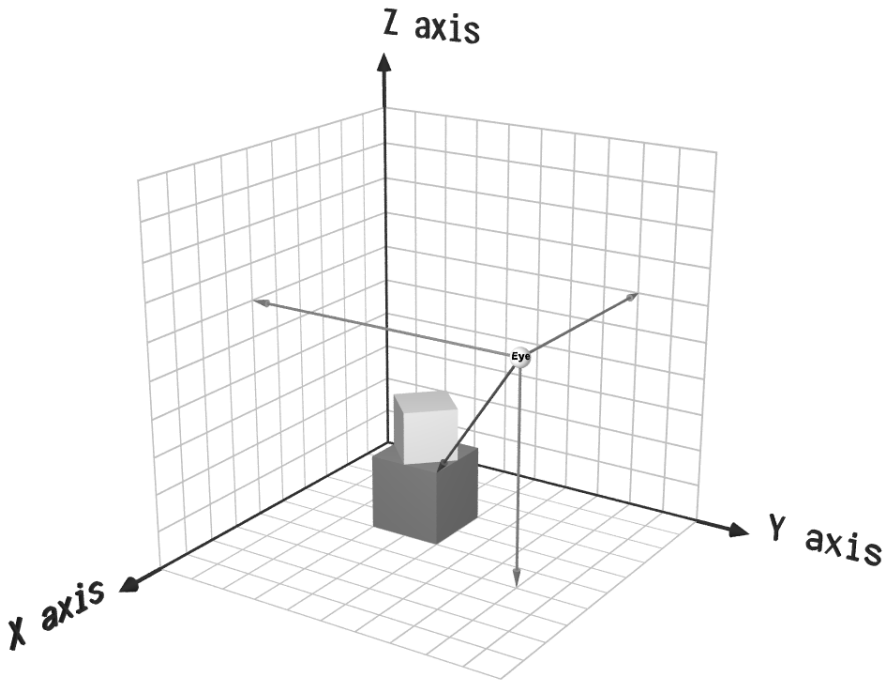
Signed _____ Date _____
(5 points)

MODELING AND VIEWING: You are given the following subset of the Matrix3D object. As a reminder; all methods compose their transformation with the current value of the matrix object to which they are applied. For example, A.*method(args)* $\rightarrow A_{new} = A_{old}\, M_{method}$.

```
public class Matrix3D {
        //
        // Constructors
        //
        public Matrix3D();                  // initializes with identity transform

        //
        // General interface methods
        //
        public void translate(float tx, float ty, float tz);
        public void scale(float sx, float sy, float sz);
        public void rotate(float ax, float ay, float ay, float angle);
        public void lookAt(float eyex, float eyey, float eyez,
                           float atx,  float aty,  float atz,
                           float upx,  float upy,  float upz);
        public void perspective(float left, float right,
                                float bottom, float top,
                                float near, float far);
}
```

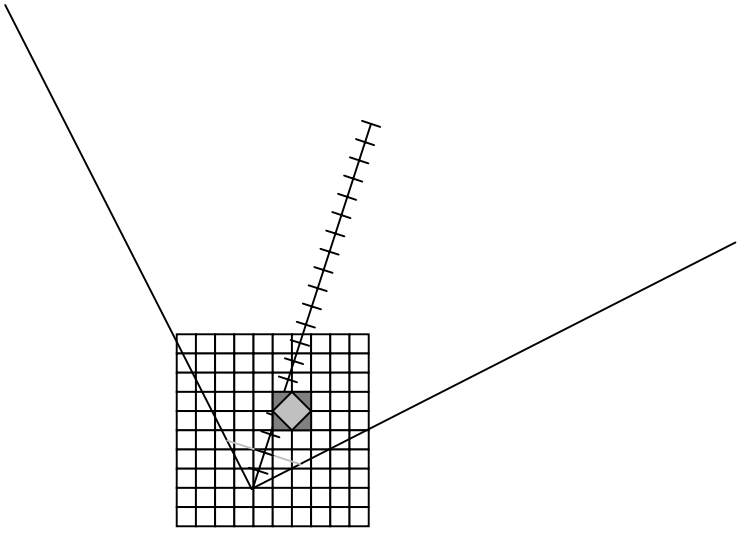
You are also given the method, `drawCube(Matrix3D M)`, that draws a cube with model-space coordinates { (-1,-1,-1), (1,-1,-1), (1,1,-1), (-1,1,-1), (-1,-1,1), (1,-1,1), (1,1,1), (-1,1,1) }.



Part A: (15 points) Write a code fragment using the methods given that draws the scene shown above. The only two objects in the scene are the gray cubes shown; all other objects are for illustration purposes only. The z-axis should project to a vertical column in the image that you render. The corner of the dark block that is indicated by the arrow should be in the center of your image. The eye position is the center of the sphere shown.

Each edge of the darker cube is 2 units. The sides of the lighter cube are $\sqrt{2}$ units. It is rotated 45 degrees, and is sitting on top of the darker cube. Choose a viewing frustum of 90 degrees both vertically and horizontally, and a range of z values that includes both cubes. Don't worry about illumination. The order of operations is most important. **(space for Part A)**

```
drawIt() {
   Matrix3D m = new Matrix3D( );
   m.perspective(-2.0f, 2.0f,  -2.0f, 2.0f,  1.0f, 20.0f);
   m.lookat(6.0f, 8.0f,6.0f,          // eye
            5.0f,5.0f,2.0f,           // lookat
            0.0f,0.0f,1.0f);          // up
   m.translate(4.0f, 4.0f, 1.0);
   drawCube( );                       // dark cube
   float s = Math.sqrt(2.0);
   m.translate(0.0f, 0.0f, 1.0f + s);
   m.rotate(0.0f, 0.0f, 1.0f, (float) (Math.PI/4.0));
   m.scale(s, s, s);
   drawCube( );                       // lighter cube
}
```



Part B: (10 points) Give a 4 by 4 matrix that rotates points about the vector $\begin{bmatrix}3 & 12 & 4\end{bmatrix}^T$ by 30 degrees counterclockwise. You can leave your result as a sum or product of matrices if you wish and there is no need to simplify expressions (Note: cos 30º = $\frac{\sqrt{3}}{2}$, sin 30º = $\frac{1}{2}$).

Normalized rotation axis: $\begin{bmatrix}\frac{3}{13} & \frac{12}{13} & \frac{4}{13}\end{bmatrix}$

Here's the easy way:

$$R = \left(1-\frac{\sqrt{3}}{2}\right)\begin{bmatrix}\frac{9}{169} & \frac{36}{169} & \frac{12}{169}\\ \frac{36}{169} & \frac{144}{169} & \frac{48}{169}\\ \frac{12}{169} & \frac{48}{169} & \frac{16}{169}\end{bmatrix} + \frac{1}{2}\begin{bmatrix}0 & -\frac{4}{13} & \frac{12}{13}\\ \frac{4}{13} & 0 & -\frac{3}{13}\\ -\frac{12}{13} & \frac{3}{13} & 0\end{bmatrix} + \frac{\sqrt{3}}{2}\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

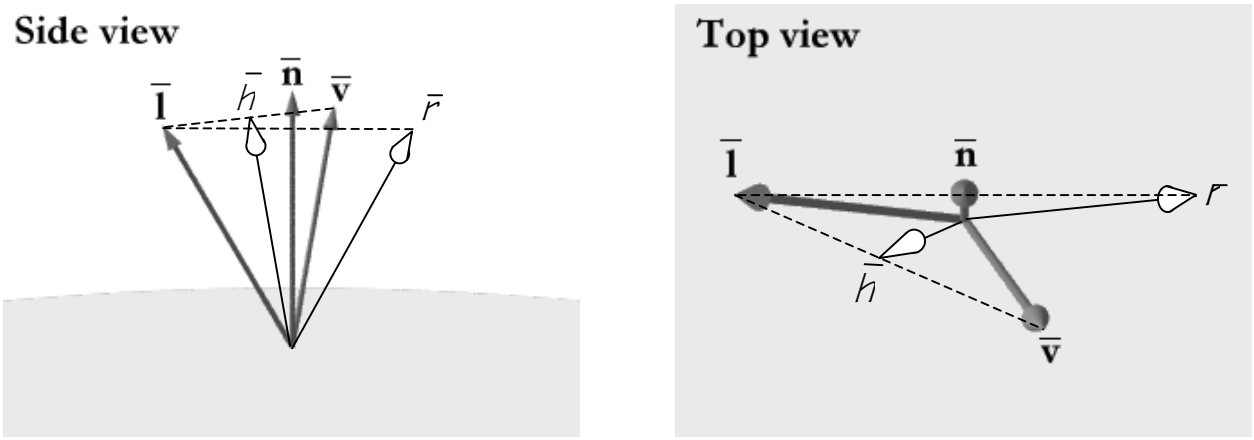
$$R = \frac{1}{338}\begin{bmatrix}18+160\sqrt{3} & 20-36\sqrt{3} & 180-12\sqrt{3}\\ 124-36\sqrt{3} & 288+25\sqrt{3} & 57-48\sqrt{3}\\ -132-12\sqrt{3} & 135-48\sqrt{3} & 32+153\sqrt{3}\end{bmatrix}$$

Inserting into a 4 by 4

$$R = \begin{bmatrix}18+160\sqrt{3} & 20-36\sqrt{3} & 180-12\sqrt{3} & 0\\ 124-36\sqrt{3} & 288+25\sqrt{3} & 57-48\sqrt{3} & 0\\ -132-12\sqrt{3} & 135-48\sqrt{3} & 32+153\sqrt{3} & 0\\ 0 & 0 & 0 & 338\end{bmatrix}$$

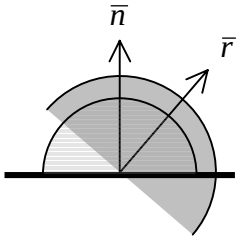
ILLUMINATION MODELS: In class we have discussed two methods for computing specular highlights. The first method, which is attributed to Phong, uses the angle between the viewing vector and the mirror reflection direction, $I_{Phong} = (\bar{v} \cdot \bar{r})^n$. The second method, proposed by Blinn, uses the angle between the surface normal and the so-called *half-way* vector that bisects the incoming light direction and the viewing direction $I_{Blinn} = (\bar{n} \cdot \bar{h})^n$.

Part A: (5 points) On the two figures shown below sketch both the mirror reflection direction, $\bar{r}$, and the halfway vector, $\bar{h}$. Which subsets of three or more vectors from the set $\{\bar{l}, \bar{n}, \bar{v}, \bar{r}, \bar{h}\}$ are coplanar?



The vectors $\bar{l}, \bar{n}$ and $\bar{r}$ are coplanar, The vectors $\bar{l}, \bar{h}$ and $\bar{v}$ are also coplanar.

Part B: (10 points) Assume that the vector to the light source remains fixed while the viewing vector is varied. Sketch and discuss the range of angles above the surface where I_{Phong} has a non-zero value and identify the direction of maximum reflection. Compare this range and the direction of maximum reflection to I_{Blinn}.



Blinn's specular model has a non-zero value for all viewing directions within the hemisphere above the surface as long as the light source is on the same side. In contrast, Phong's model only has a non-zero value for the hemisphere surrounding the reflection direction,, part of which is always below the surface. Both Blinn's and Phongs specular models reach a maximum value along the reflected direction $\bar{r}$.

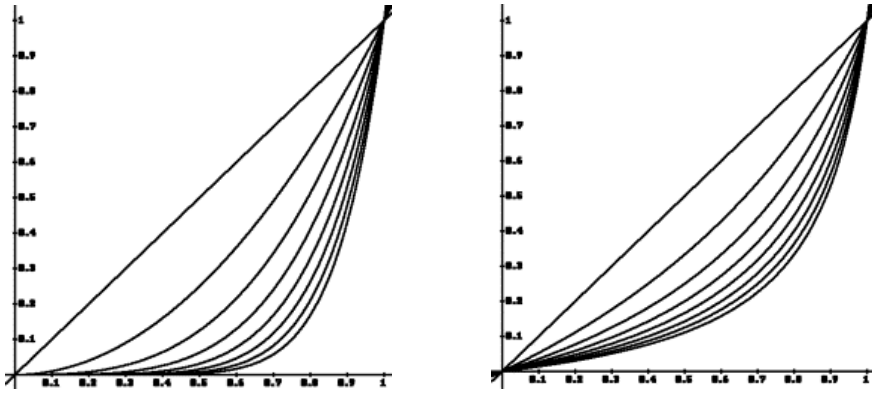
Part C: (10 points) Since the shininess exponent, n, used by both the Phong and Blinn specular illumination models is empirical rather than physically based, throughout the years several alternatives have been suggested. The basic form of both the Phong and Blinn models is

$$I_{specular} = f(\bar{n}, \bar{l}, \bar{v})^n \text{ where } 0 \le f(\bar{n}, \bar{l}, \bar{v}) \le 1$$

Consider the following alternative model

$$I_{specular} = \frac{f(\bar{n}, \bar{l}, \bar{v})}{n - nf(\bar{n}, \bar{l}, \bar{v}) + f(\bar{n}, \bar{l}, \bar{v})}$$

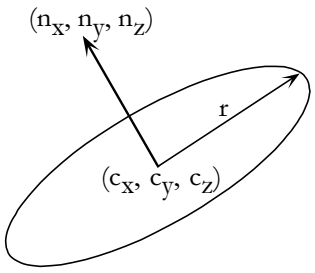
Make rough sketches of both the standard and alternative models over the valid range of $f(\bar{n}, \bar{l}, \bar{v})$, for values of *n* equal to 1, 2, 4, 8, and any others you'd like (Hint: simply treat $f(\bar{n}, \bar{l}, \bar{v})$ as a variable between 0 and 1). How does this new model compare to the standard specular reflection model? What possible advantages might this new model have?



The standard shininess function, shown on the left, attenuates the specular spot size faster than alternative model and its fall off is more gradual. Thus, a larger value of n is needed to create a similar effect. Otherwise, the behavior of the two functions is quite similar.

Since, neither model has any physical basis the alternative model could be used in place of the standard one with appropriate adjustments in n values. It has the advantage that it can be computed using a multiply, a subtraction, an addition, and a divide compared to a pow( ) function for the standard model, which involves a log, a multiply, and an antilog.

RAY TRACING: Consider a circular disc which is defined by the coordinate of it's center, a direction normal to the disk, and a radius.



All points, $\begin{bmatrix} x & y & z \end{bmatrix}^T$, on the disk satisfy the following plane equation:

$$n_x x + n_y y + n_z z = n_x c_x + n_y c_y + n_z c_y$$

as well as the following distance constraint:

$$(c_x - x)^2 + (c_y - y)^2 + (c_z - z)^2 \le r^2$$

You are given the following classes

```
class Vector3D {
  public float x, y, z;

  // constructors
  public Vector3D( );
  public Vector3D(float x, float y, float z);

  // methods
  public static float dot(Vector3D A, Vector3D B);
  public static Vector3D cross(Vector3D A, Vector3D B);
  public static Vector3D normalize(Vector3D A);
  public static Vector3D scale(Vector3D A, float s);
  public static Vector3D plus(Vector3D A, Vector3D B);
  public static Vector3D minus(Vector3D A, Vector3D B);
}

class Ray {
  public static final float MAX_T = Float.MAX_VALUE;
  public Vector3D origin;
  public Vector3D direction;
  public float t;
  public Renderable object;
        :
}

class Disc implements Renderable {
  Surface surface;
  Vector3D center;
  Vector3D normal;
  float radius;

  public Disc(Surface s, Vector3D c, Vector3D n, float r) {
    surface = s;    center = c;    normal = Vector3D.normalize(n);    radius = r;
  }

  public boolean intersect(Ray ray) { // returns false if ray does not intersect
      // write your code here
  }
  :
}
```

Part A: (10 points) Write a code fragment for the intersect( ) method of the Disc object. The method should return true in the case of an intersection and update the relevant fields of the Ray object that it is passed. Otherwise, it should return false. Also, assume that a ray can intersect the disc from either side.

```
public boolean intersect(Ray ray) {
    // check distance constraint first
    Vector3D x = center.minus(ray.origin);
    float t = Vector3D.dot(x, ray.direction);
    if (Vector3D.dot(x, x) - t*t >= radius*radius)
        return false;

    // find intersection with plane
    t = Vector3D.dot(normal, ray.direction);
    if (t == 0)
        return false;          // ray is parallel to disc

    t = Vector3D.dot(normal, x) / t;
    if ((t < 0) || (t > ray.t))
        return false;

    x = x.minus(ray.direction.scale(t));
    if (Vector3D.dot(x, x) >= radius*radius)
        return false;

    ray.t = t;
    ray.object = this;
    return true;
}


public boolean intersect(Ray ray){
    // compute intersection with plane
    float t = Vector3D.dot(normal, ray.direction);
    if (t == 0)
        return false;          // ray is parallel to disc

    Vector3D x = center.minus(ray.origin);
    t = Vector3D.dot(normal, x) / t;
    if ((t < 0) || (t > ray.t))
        return false;

    // then compute distance constraint
    x = x.minus(ray.direction.scale(t));
    if (Vector3D.dot(x, x) >= radius*radius)
        return false;

    ray.t = t;
    ray.object = this;
    return true;
  }
```

Part B: (5 points) What pre-computation could have been done, and / or instance variables could have been added to the Disc() constructor that would have reduced the amount of work done in the intersect() method?

The following values could be computed once by the constructor and reused on each call to the intersect method:

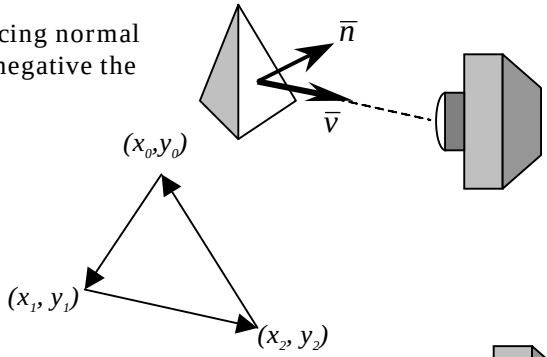
```
float radSqr = radius * radius;
float nDotC = Vector3D.dot(normal, center);
```

Part C: (10 points) When you were implementing the disc intersection test you had the choice of testing for intersection with the infinite plane, $n_x x + n_y y + n_z z = n_x c_x + n_y c_y + n_z c_y$, or testing the distance constraint, $(c_x - x)^2 + (c_y - y)^2 + (c_z - z)^2 \le r^2$, first. Discuss the relative merits of each approach.

Computing the intersection with the infinite plane first leads to simpler and shorter code. However, by testing the distance constraint first it is possible to more reject rays sooner, and thus the code is faster. The reason that testing the distance constraint first rejects more rays is that, in general, fewer rays pass within of a given distance of a point than intersect an infinite ray. The exception occurs when the ray's origin is within a disc radius from the disc center. Code for both versions was given for part a. In my implementation the testing the distance constraint first is about 20% faster than finding the plane intersection first.

VISIBLE-SURFACE DETERMINATION: Back-face culling is one of the simplest and efficient methods for determining the visibility of a polygonal facet. It is frequently used as a visibility-preprocessing step in conjunction with other visibility techniques such as depth buffering. Below three different techniques for back-face culling are described.

Method 1: Compute the dot product of each facet's outward facing normal with a vector from the surface towards the eye. If this value is negative the surface is back facing and can be culled.

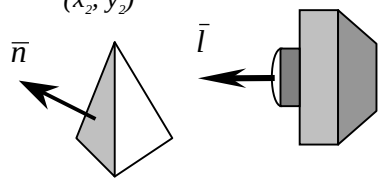


Method 2: If the screen-space area of the polygon as given by the formula

$$A = \tfrac{1}{2}\sum_{i=1}^{n}(x_{i \bmod n} - x_{i-1 \bmod n})(y_{i \bmod n} + y_{i-1 \bmod n})$$

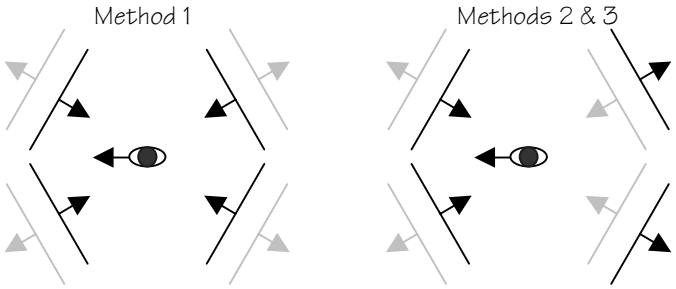
is negative the surface is back facing and can be culled.

Method 3: As described in Hearns and Baker (pp. 471-472), compute the dot product between the surface normal and a vector along the viewing direction. If it is positive the facet is back facing and can be culled.



Part A: (10 points) Do all three approaches give identical results? For each method explain which space or which part of the rendering pipeline it is best suited for. What are the relative advantages and disadvantages of each method?

All three methods yield the same results for facets in front of the camera. However, the first method differs from the other two for facets behind the viewpoint. Facets behind the viewpoint are generally irrelevant for Method 2 since it is applied after clipping. However, these facets behind the viewpoint will be considered by methods 1 and 3.



Method 1 could be used in any of modeling, world, or eye spaces. Method 2 is best suited for screen space. Method 3, like 1, could be used in any of modeling, world, or eye spaces, but it is probably best suited for eye space since the viewing direction there is along the z-axis, thus simplifying the dot product computation.

The advantage of Methods 1 and 3 are that it can be applied prior to any illumination calculations, whereas Method 2 occurs after projection, which is much later in the pipeline.

Part B: (5 points) What restrictions must be satisfied for a model to be properly rendered using back-face culling followed by depth buffering? What additional restrictions must be satisfied if *only* back-face culling is used to determine visibility?

When both back-face culling and depth buffering are used the object must be a closed" watertight" model (i.e. there can be no holes that let you see inside of the object) and its facet vertices must be described consistently in either clockwise or counter-clockwise orders. If only back-face culling is used to resolve visibility then the object must also be convex.

Part C: (5 points) Does a combination of any 2 or all 3 of the back-face culling methods given provide an additional advantage? Estimate an upper bound on the number of facets that can be removed using these methods alone and in combination.

The combination of methods 1 and 3 can provide a significant advantage of removing all facets that are behind the viewpoint and thus unseen. Used alone all methods can reduce the number of facets considered by 50%. The combination of 1 and 3 might eliminate as much as 75% of the facets.