# Homework #2

## Projections, Hidden Surfaces, Shading, Ray Tracing, Texture Mapping, and Parametric Curves

**Prepared by: Luke Meyers, Ian Li, and Zoran Popović**

**Assigned: Friday, November 9, 2001**
**Due: Wednesday, November 28, 2001**

**Directions: Provide short written answers to the questions in the space provided. If you require extra space, you may staple additional pages to the back of your assignment. Feel free to talk over the problems with classmates, but please answer the questions on you own.**

Name: _____

## Problem 1. Projections (9 Points)

Perspective Projections

*True or False*

1. Size varies inversely with distance

2. Distance and angles are preserved

3. Parallel lines do not remain parallel

4. Perspective projections make z-buffers more imprecise than orthographic projections

Parallel Projections

*True or False*

5. More realistic looking than perspective projections
6. Good for exact measurements

7. Parallel lines do not remain parallel

8. Angles (in general) are not preserved

9. Lengths vary with distance to the eye

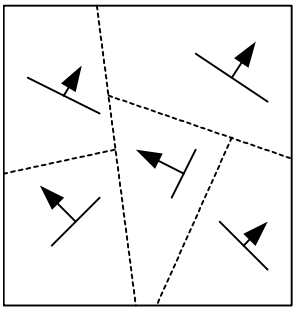
## Problem 2. Cabinet Projection (5 Points)

Cabinet projection is an oblique projection for which edges perpendicular to the plane of projection are projected to one-half of their original size. Show that for a cabinet projection the angle between the direction of projection and the plane of projection is 63.43 degrees.

## Problem 3. BSP Trees (20 Points)

BSP trees are widely used in computer graphics. Many variations can be used to increase performance. The following questions deal with some of these variations.

For the version of BSP trees that we learned about in class, polygons in the scene (or more precisely, their supporting planes) were used to do the scene splitting. However, it is not necessary to use existing polygons – one can choose arbitrary planes to split the scene:



a. What is one advantage of being able to pick the plane used to divide the scene at each step? What is one disadvantage of not just using existing polygons?

Recall that when using a BSP tree as described in class, we must draw *all* the polygons in the tree. This is very inefficient, since many of these polygons will be completely outside of the view frustum. However, it is possible to store information at the internal nodes in a BSP tree that will allow us to easily determine if any of the polygons below that node will be visible. If none of the polygons in that sub-tree will be visible, we can completely ignore that branch of the tree.

b. Explain what extra information should be stored at the internal nodes to allow this, and how it would be used to do this "pruning" of the BSP tree.

## BSP Trees (cont'd)

c. In class, we talked about doing a "back to front" traversal of a BSP tree. But it is sometimes preferable to do a "front to back" traversal of the tree, in which we draw polygons closer to the viewer before we draw the polygons farther away. (See part (d) for one reason why this is useful) How should the tree traversal order be changed in order to do a front to back traversal?

When we traverse a BSP tree in back to front order, we may draw over the same pixel location many times, which is inefficient since we would do a lot of "useless" shading computations. Assume we instead traverse the tree in front to back order. As we scan convert each polygon, we would like to be able to know whether or not each pixel of it will be visible in the final scene (and thus whether we need to compute shading information for that point).

d. What simple information about the screen do we need to maintain in order to know if each pixel in the next polygon we draw will be visible or not?

## Problem 4. Ray Tracing (Z-Buffer and Distribution Ray Tracing) (20 points)

Suppose we want to combine the Z-buffer algorithm with ray tracing (assuming that the scene consists only of polygons). We can assign to each polygon a unique emissive color corresponding to an "object ID". Then, we turn off all lighting and render the scene from a given point of view. At each pixel, the Z-buffer now contains the object point (indicated by the pixel x,y coords and the z-value stored in the buffer) and an object ID which we can look up to figure out the shading parameters. In effect, we have done a ray cast for a set of rays passing through a given point (the center or projection).

a. Consider the figure below. The ray cast from the eye point through an $n$ x $n$ image (projection) plane is actually going to be computed using the Z-buffer algorithm described above. In effect, how many rays are fired from the eye to determine the first intersected surfaces?

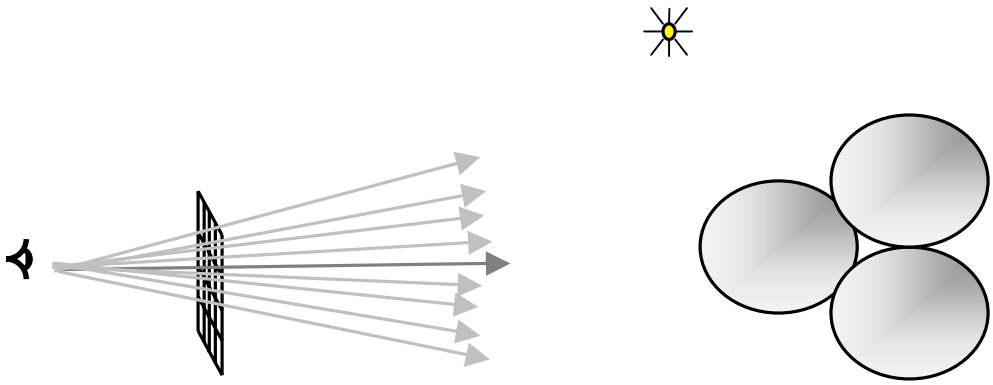
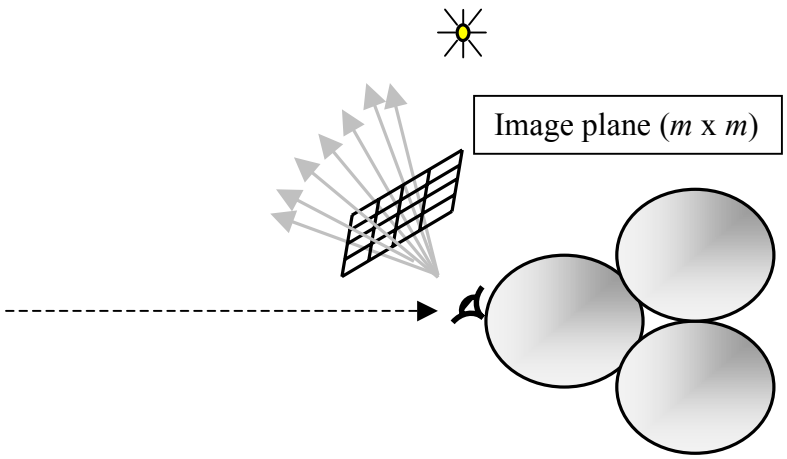


Image plane ($n$ x $n$)

b. Now let's say we want to capture glossy reflections by spawning many rays at each intersection point roughly in the specular direction. As indicated by the figure below, we can cast these rays by positioning the new viewpoint at a given intersection point, setting up a new image plane of size $m$ x $m$ oriented to align with the specular direction, and then run our modified Z-buffer algorithm again. Let's say we follow this procedure for all pixels for $k$ bounces in a scene assuming non-refractive surfaces. In effect, how many rays will we end up tracing?



## Z-Buffer and Distribution Ray Tracing (cont'd)

c. If we compute a spread of transmitted rays to simulate translucency as well, how many rays will we end up tracing?

Now, instead of using z buffer algorithm we use the distribution ray tracing method. Count how many rays we need to trace. For each of these cases:

d. First intersections (0 bounces)

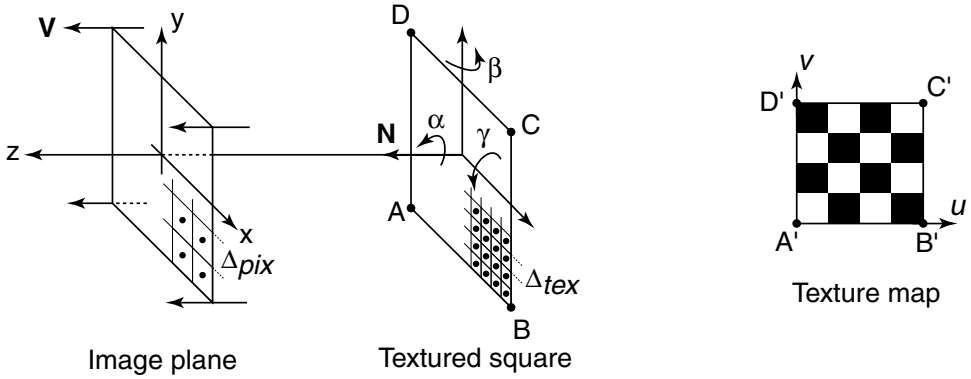
e. *k* bounces to simulate gloss only (no refraction/translucency).

f. *k* bounces to simulate both gloss and translucency.

g. Given your answers to (a)-(f) when would it be appropriate to use the modified Z-buffer in a ray tracing algorithm?

## Problem 5. Texture filtering (20 points)

In class, we discussed how brute forces sampling, mip maps, and summed area tables can be employed to anti-alias textures. The latter two techniques average over a region of the texture image very quickly with varying degrees of accuracy, which we consider further in this problem. Consider the scene below: an ***orthographic viewer*** looking down the –z-axis views a textured square. The image size and square size are the same and they are initially aligned to one another as shown. The pixel spacing on the image plane and the texel spacing on the square are $\Delta_{pix}$ and $\Delta_{tex}$, respectively.




a. Assuming $\Delta_{pix} > \Delta_{tex}$, how must these sample spacings be related in order for mip mapping to yield the correct values without interpolating among mip map levels?

b. Consider the coordinate system of the square shown in terms of the normal **N** and the two axes aligned with the x and y axes in the figure. Assume that we have the freedom to rotate the square about any *one* of those local axes, as indicated by rotation angles $\alpha$, $\beta$, and $\gamma$. What restriction do we have on rotation about any one of these axes in order for mip mapping to return the correct average texture values? [For example, you could decide that $\alpha$, $\beta$, and $\gamma$ must all be zero degrees, or you could decide that some of them can vary freely, or you can decide that some can take on a set of specific values. Do not focus on rotations that cause the square to be back-facing.]

## Texture filtering (cont'd)

c. Now assume we start again with the unrotated geometry and that we're using summed area tables. If linear interpolation within the summed area table causes no significant degradation, what restriction, if any, should we place on the relative pixel and texel spacings to get correct texture averaging?

d. As in (b), what restriction must we place on rotation about any one of the given axes in order for summed area tables to return the correct average texture values?

## Problem 6. Parametric Curves (11 Points)

Bezier curves are very simple and so versatile that almost all graphics packages feature a Bezier curve tool. Now suppose you bought **BezierDraw**, a graphics program that only has a Bezier curve tool and nothing else. (Assume that all curves that **BezierDraw** creates are third-order Beziers.)

a. Is it possible to draw a perfect circle with **BezierDraw**? Explain why or why not. (Hint: what's the parametric formulation of a circle?)

b. Suppose two other graphics programs were released: **C2Draw**, a program that features only C2-interpolating curves, and **CatmullRomDraw**, a program that draws using Catmull-Rom curves. Describe in detail what the advantages and disadvantages are of each of the 4 graphics programs.

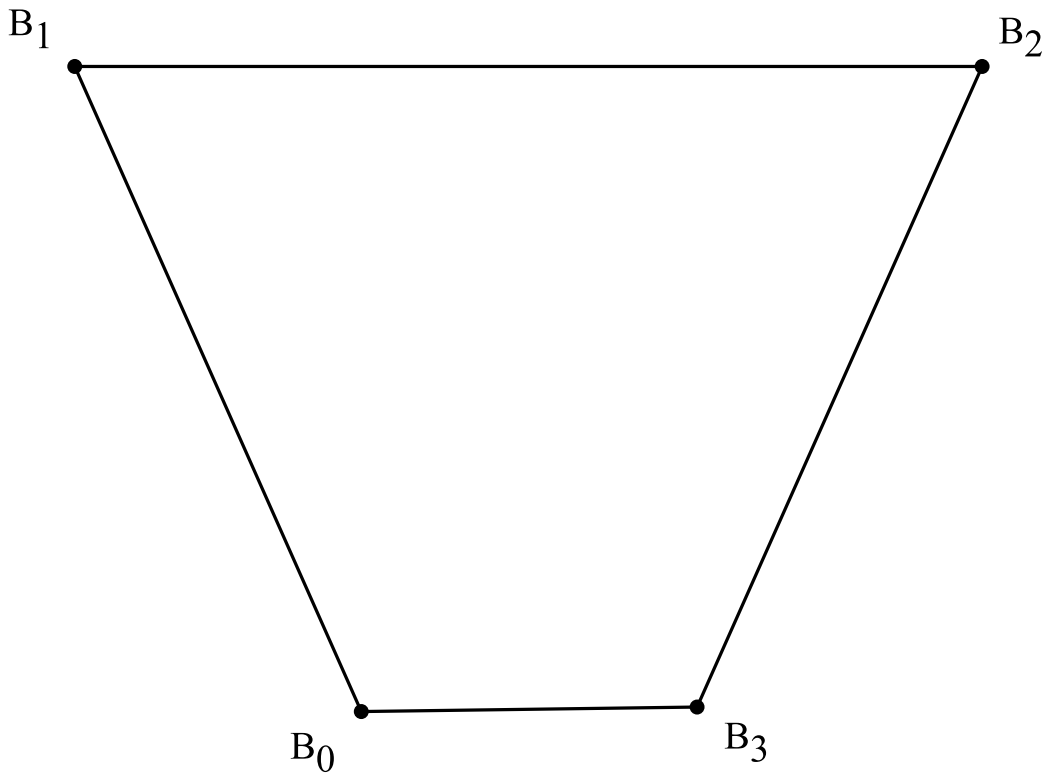
## Problem 7. Constructing Splines (20 points)

By connecting a sequence of Bézier curves together, we can construct spline curves such as B-splines and Catmull-Rom splines.

In this problem, you will construct Bézier control points and sketch curves. You only need to get the lengths of line segments approximately right, and you need only label the diagrams as requested in the problem statements.

a) **(5 points)** Consider a **closed-loop cubic B-spline** curve with control points, $B_0$, $B_1$, $B_2$, and $B_3$.

- Construct all of the Bézier points generated by these control points, and label them $T_0$, … $T_3$, then $U_0$, … $U_3$, etc.
- Sketch the curve (just an approximate sketch that suggests the shape is sufficient).

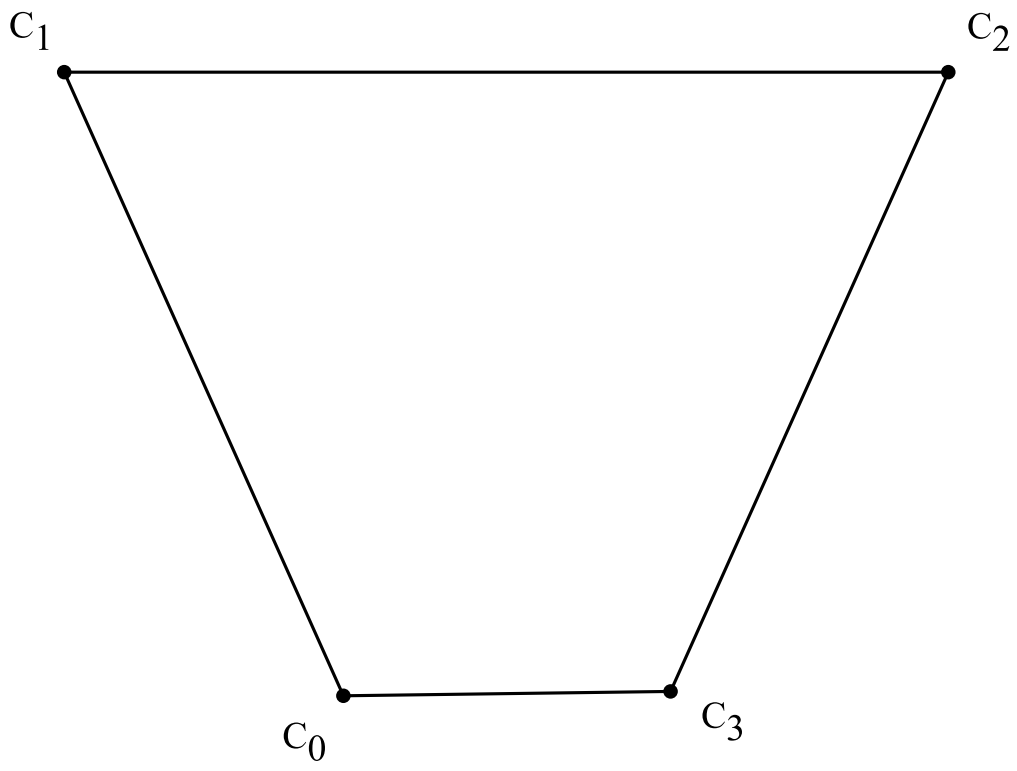



b) **(2 points)** If you move one control point in your sketch, will the whole curve change? Justify your answer.

## Constructing Splines (cont'd)

c) **(5 points)** Consider a **closed-loop Catmull-Rom** curve with control points, $C_0$, $C_1$, $C_2$, and $C_3$ and default tension, $\tau = 1/2$.

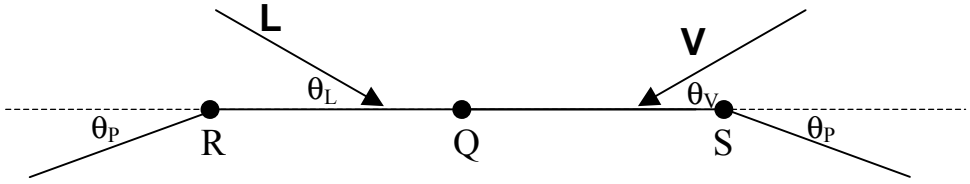
- Construct all of the Bézier points generated by these control points, and label them $T_0$, … $T_3$, then $U_0$, … $U_3$, etc.
- Sketch the rest of the curve (just an approximate sketch that suggests the shape is sufficient).




d) **(2 points)** If you move one control point in your sketch, will the whole curve change? Justify your answer.

## Problem 8. Polygon Shading (20 Points)

Typically, when shading polygons, speed is at least as important a consideration as appearance. Two popular shading models, Phong interpolation and Gouraud interpolation, differ in these respects. Phong interpolation (not to be confused with the Phong lighting model, a separate but also-important contrubution of Mr. Wu Tong Phong) involves interpolating the normals along the surface of each polygon, and is consequently slower to compute than Gouraud, which simply computes the normals at each vertex and linearly interpolates the colors themselves. As we shall see, Gouraud interpolation has some drawbacks.




$\theta_L = 30°$
$\theta_V = 30°$
$\theta_P = 20°$

To simplify matters, we will consider a 2-dimensional case. The computations are easier, but the same principles apply in three dimensions. The diagram above portrays a line $\overline{RS}$ that we wish to shade, along with its two neighbors. The (unit) vectors **L** and **V** represent the angles of incident light (from a single directional source) and the viewing angle respectively. Assume the viewer to be far enough away that $\theta_V$ is equal everywhere.

a. Use Phong interpolation to find the intensity of light (as an RGB value) observed at point Q (halfway between R and S). Use the following simplified version of the Phong lighting model:

$$I = I_L [ k_d (\mathbf{N} \bullet \mathbf{L})_+ + k_s (\mathbf{V} \bullet \mathbf{R})_+^{n_s} ]$$

Use the values $I_L = (100_R, 100_G, 100_B)$, $k_d = (0.2_R, 1.0_G, 0.2_B)$, $k_s = (1.0_R, 1.0_G, 1.0_B)$, and $n_s = 100$.

## Polygon Shading (cont'd)

b. Does this intensity include a significant contribution from specular highlight? How can you tell?

c. Compute the intensity at Q again, but this time use Gouraud interpolation. That means you'll have to find the color at points R and S, and interpolate linearly to find it at Q. Use the same simplified Phong lighting model as in part a. Show your work.

d. Does the intensity computed via Gouraud interpolation include a significant contribution from specular highlight? How can you tell? How does this differ from the result seen with Phong interpolation? Explain why this is so, and what differences might be seen between the two in other situations.

## Polygon Shading (cont'd)

e. A straightforward implementation of Phong interpolation is impractical for realtime rendering on current hardware. Some enterprising individuals have endeavored to devise approximations that achieve the benefits of Phong interpolation without the expense of all that computation (to give you an idea, a straightforward implementation requires something like 4 multiplies, 2 adds, a divide, and one square root per pixel).

One proposed method begins with the approximation of replacing $\mathbf{V \bullet R}$ in the Phong lighting equation with $\mathbf{N \bullet L}$. This means that the specular highlight will be maximal from any viewing direction (though it is still affected by the angle between the light and the surface). The resulting equation is this:

$$I = I_L \left[ k_d \, (\mathbf{N \bullet L})_+ + k_s \, (\mathbf{N \bullet L})_+^{n_s} \right]$$

Note that $(\mathbf{N \bullet L})_+$ is just $\sin(\theta_L)$, so we can rewrite the above as:

$$I = I_L \left[ k_d \sin(\theta_L) + k_s \sin(\theta_L)^{n_s} \right]$$

The above equation is quite efficient to interpolate – rather than interpolating normal vectors, all we must do is interpolate the angle $\theta_L$ at each point. The sine can be efficiently computed via a lookup table.

Or so the argument goes. Give this approximation some thought and discuss it with some classmates if possible. Does it produce a satisfactory approximation of Phong interpolation model? In particular, would this approximation avoid the problem with specular highlights that you discovered about Gouraud interpolation in part d of this problem?