Homework #2

Assigned: May 5, 2003
Due: May 20, 2003
(collection box outside 3561)

Directions

Please 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 your own.

Name: Key

Problem 1. Color

Why is the sky blue?

You may already know that sunlight is made up of all the colors of the rainbow: red, orange, yellow, green, blue, and violet. You probably also know that sunlight has to pass through our atmosphere before it reaches our eyes. The gas molecules in the atmosphere break up, or "scatter," the sunlight into its many parts. But they scatter some parts more effectively than others.

Different colors of light have different energies, or wavelengths. Red light has a long wavelength and a lower energy; blue light has a short wavelength and a higher energy. The gas molecules in the atmosphere scatter the higher-energy blue wavelengths better than the red wavelengths. So the sky looks blue.

Problem 2. Shading

The full moon looks like a flat, uniformly illuminated disc. However, we know that it is (approximately) spherical, and the irradiance (power per unit area) from the sun falls off as we go from the surface directly facing towards the sun around to the terminator. In class, we learned that it falls off as a cosine function. Therefore the brightness (a.k.a. radiance) should likewise fall off as a cosine from the center towards the edge. Like a lambertian sphere when lit, the moon should look brightest at the center, gradually becoming dimmer at the edge. Is the Moon not a lambertian (diffuse) reflector? Apparently not.

Why does the full moon look flat?

We know the moon is a sphere, so we expect the center of the moon be brighter, and that the side or silhouette of it be darker. What is wrong?

The shading of a Lambertian surface is

$$E = I_i \, a \cos \theta_i$$

where I_i is the incident light, a is the reflectance coefficient, or "albedo" which contributes to the ink blotches on the moon, and $\cos \theta_i$ is the angle L •N, where L is the direction of the light source and N is the normal. Note that we constrain $\cos \theta_i$ between 0 to 1.

We are used to Lambertian surface, so we get used to perceive shape information (i.e., N) from this equation. If we see constant brightness, we will regard the surface as flat, i.e., N is constant.

However, the material on the moon does NOT obey Lambertian law, so our habit gives us wrong perception. Actually, the material on the moon has the following property: The more oblique the viewing direction, the brighter it will look, i.e.,

$$E = I_i \, a \cos \theta_i / \cos \theta_e$$

If θ_e increases, $1/\cos \theta_e$ increases, i.e., θ_e increases, then E also increases.

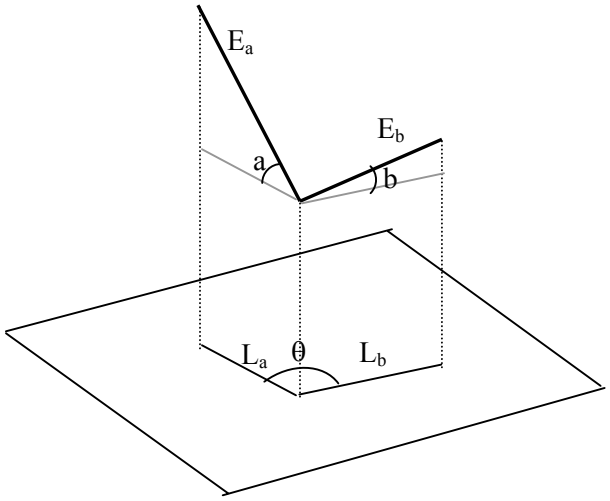
For a full moon: $\theta_e = \theta_i$, So $E = I_i \, a \, 1 = I_i \, a$.

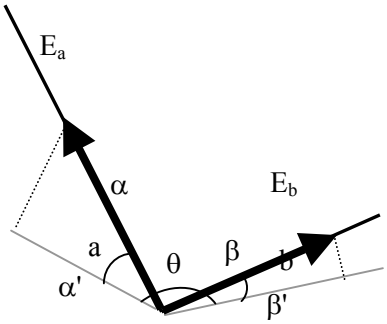
Thus, we see uniform brightness regardless of shape. Here, a is the different reflectance ratio of different regions of the moon, which explains why we see some dark regions on the moon.

Problem 3. Orthographic projection

Consider two lines E_a and E_b in a plane that are at right angles to each other. Suppose E_a and E_b are projected onto an image plane (not necessarily parallel to the plane of the lines) by an orthographic projection, as shown in the figure below. Let L_a and L_b be the respective projected lines of E_a and E_b. Show that the angle subtended by L_a and L_b (say θ) is related to that by E_a and L_a (say a) and that by E_b and L_b (say b) by the following:

$$\cos\theta = -\tan a \tan b$$





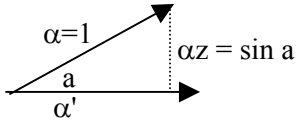
Let's move the image plane until it touches the corner. For simplicity, we select this point as the origin. Take the unit vector along Ea and Eb and project them onto the image plane using orthographic projection:

α' = 1 cos a = cos a
β' = 1 cos b = cos b

α' • β' = |α'| | β'| cos θ = (cos a) (cos b) cos θ

but we also know that

α' • β' = (αx, αy, 0) • (βx, βy, 0) = (αx, αy, αz) • (βx, βy, βz) - αzβz = α • β - αzβz
= 0 - αzβz (since Ea is perpendicular to Eb)
= - αzβz
= - (sin a) (sin b)

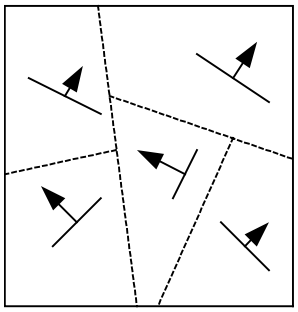


i.e., (cos a)(cos b) cos θ = -(sin a) (sin b)
cos θ = - (tan a)(tan b)

Problem 4. BSP Trees

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?

Some advantages: one is able to decrease the number of divided polygons. All of the polygons end up being leaf nodes in the BSP, thus making removal easier.

Some disadvantages: it takes time to figure out where the splitting panes should be put. It takes more space to store the larger BSP.

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.

Store the normal of the plane in the BSP. If the normal of the plane is within the viewing frustrum and the viewer is on the front side of the plane, nothing behind the plane must be evaluated.

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?

Instead of drawing everything on the other side (from the viewer) of the polygon, then the polygon, then everything on the same side (as the viewer) of the polygon, one would draw everything on the same side (as the viewer) of the polygon, then the polygon, then everything on the other side (from the viewer) of the polygon.

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?

One needs to keep track of which pixels have been drawn to and which have not been. If a pixel has not been drawn to, then draw that pixel, but if it has already been drawn to, do not draw the pixel.

Problem 5. Ray Tracing (Z-Buffer and Distribution Ray Tracing)

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 coordinates 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 of 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?

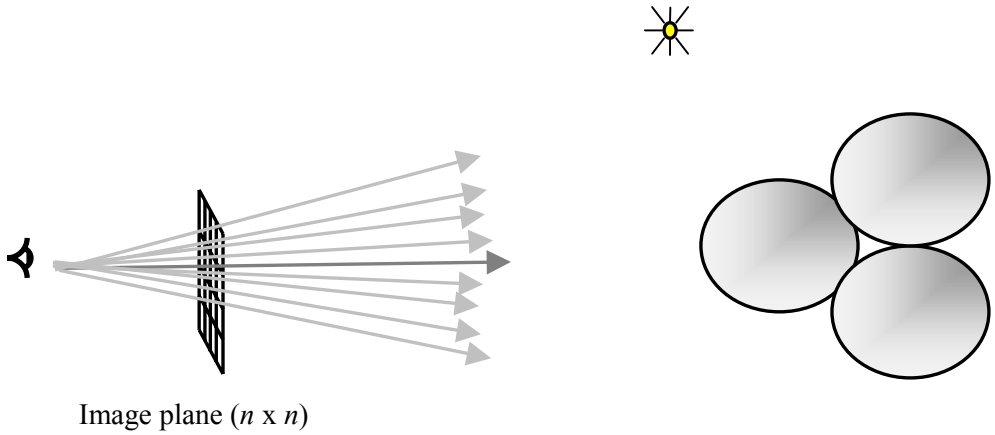


Image plane (n x n)

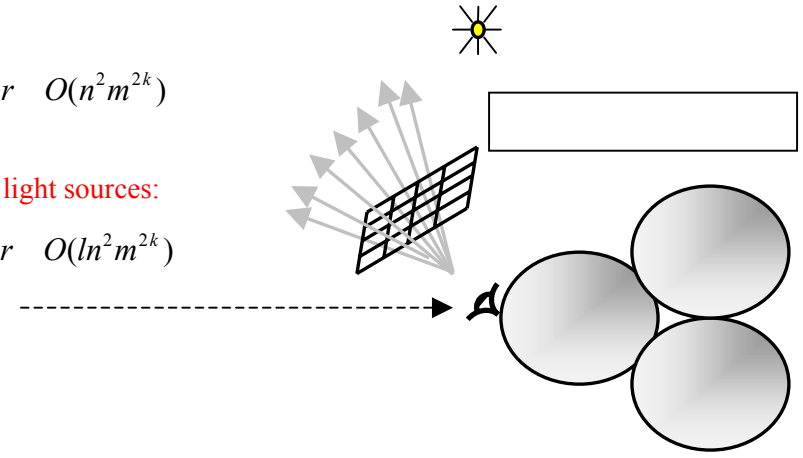
n^2 (or $16n^2$ including 4x4 antialiasing, which we will assume for this problem.)

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?

$$16n^2\sum_{i=0}^{k}(m^2)^i \quad or \quad O(n^2m^{2k})$$

Better, assuming l light sources:

$$16n^2(1+l)\sum_{i=0}^{k}(m^2)^i \quad or \quad O(ln^2m^{2k})$$



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?

$$16n^2\sum_{i=0}^{k}(2m^2)^i \quad or \quad O(n^2m^{2k})$$

Assuming l light sources:

$$16n^2(1+l)\sum_{i=0}^{k}(2m^2)^i \quad or \quad O(ln^2m^{2k})$$

Now, instead of using z buffer algorithm (i.e., we no longer use a m x m image plane), we use the distribution ray tracing method. Count how many rays we need to trace in terms of k (if necessary). For each of these cases:

d. First intersections (0 bounces)

n^2

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

Assuming 16 rays per pixel: $16kn^2$ or $O(kn^2)$
Also, assuming l light sources: $16kn^2(1+l)$ or $O(kln^2)$

f. k bounces to simulate both gloss and translucency.

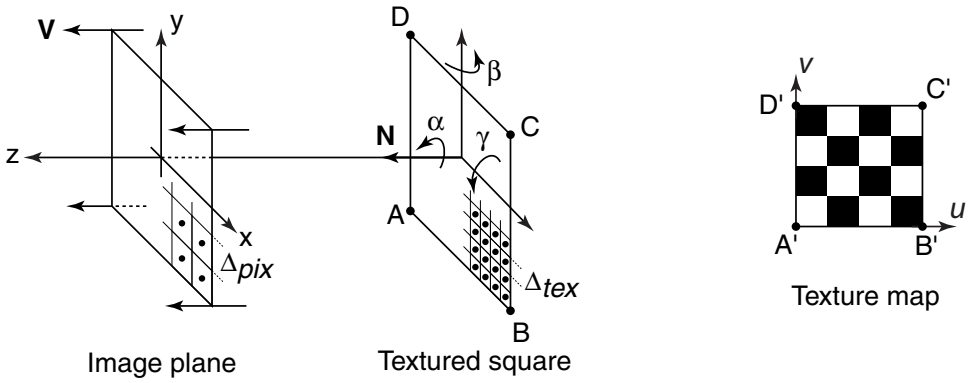
$16(2^kn^2)$ or $O(2^kn^2)$
Assuming l light sources: $16(2^kn^2)(1+ l)$ or $O(2^kln^2)$

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

Note that distribution ray tracing is essentially super-sampled Whitted ray tracing with some perturbation to each reflection and each refraction ray. The perturbation is controlled in a special way, but the ray count is exactly the same as if you ran a Whitted ray tracer with multiple rays (say 16) per pixel for antialiasing.

Problem 6. Texture filtering

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 Δ_{pix} and Δ_{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?

We want a pixel to cover a square region in the texture map of a size that fits right into one of the pyramid levels. Each pyramid level stores values that are averages over 2^i texels for the i-th level. Thus, we require:
$\Delta_{pix} = 2^i \Delta_{tex}$

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 α, β, and γ. 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 α, β, and γ 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.]

In order to get an accurate integral (and thus avoid introducing excessive blurring), a pixel would have to map onto an axis-aligned square region within the mipmap pyramid. As we tilt by β, a pixel maps onto a region that increases in width without increasing in height and is thus non-square. As we tilt by γ, a pixel maps onto a region that increases in height without increasing in width and is also non-square. Finally, as we rotate by α, a pixel maps onto a rotated square in texture space that is not axis aligned. However, rotations by α of 0, 90, 180, and 270 degrees **will** yield squares in texture space.
In summary, arbitrary rotations by α, β, and γ will yield incorrect (blurry) results, while rotations by α of 0, 90, 180, and 270 degrees will yield correct results.

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?

Each pixel projects onto an axis-aligned square in texture space. Summed area tables can integrate exactly over **any** axis-aligned square (or rectangle for that matter) accurately. Thus, there are no restrictions on pixel size.

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?

As discussed in part b, as we rotate by β and γ, the mapping of a pixel into texture space is not square, but it **is** rectangular. Thus, we can freely rotate about either one of these axes and the summed area table will give an accurate integral over the rectangular regions. However, as we rotate by α, we get a rotated square in texture space, which is not axis-aligned except when α is 0, 90, 180, or 270 degrees.

Thus, summed area tables will return the correct answer regardless of rotations by β or γ, but α must be restricted to 0, 90, 180, or 270 degrees.

Problem 7. Parametric Curves

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 or cubic 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?)

No. You can approximate a perfect circle using Bezier curves, but you wouldn't be able to draw a perfect circle. The reason is that Bezier curves are defined parametrically as sums of polynomial functions. A circle is defined parametrically as Q(t) = [cos(t) sin(t)]. cos(t) and sin(t) are not polynomial functions, therefore you can never draw a circle using Bezier curves.

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.

BsplineDraw is better than BezierDraw, because B-splines feature local control and C2-continuity. C2Draw ups the ante, by allowing the curve to interpolate the control points, while maintaining C2-continuity. Unfortunately, in C2Draw, we lose the local control we've gained with BsplineDraw. With CatmullRomDraw, our curves will interpolate control points and feature local control, but we lose C2-continuity. With both BezierDraw and BsplineDraw, the curves are within the convex hull of the control points, but C2Draw and CatmullRomDraw, we lose this nice property.

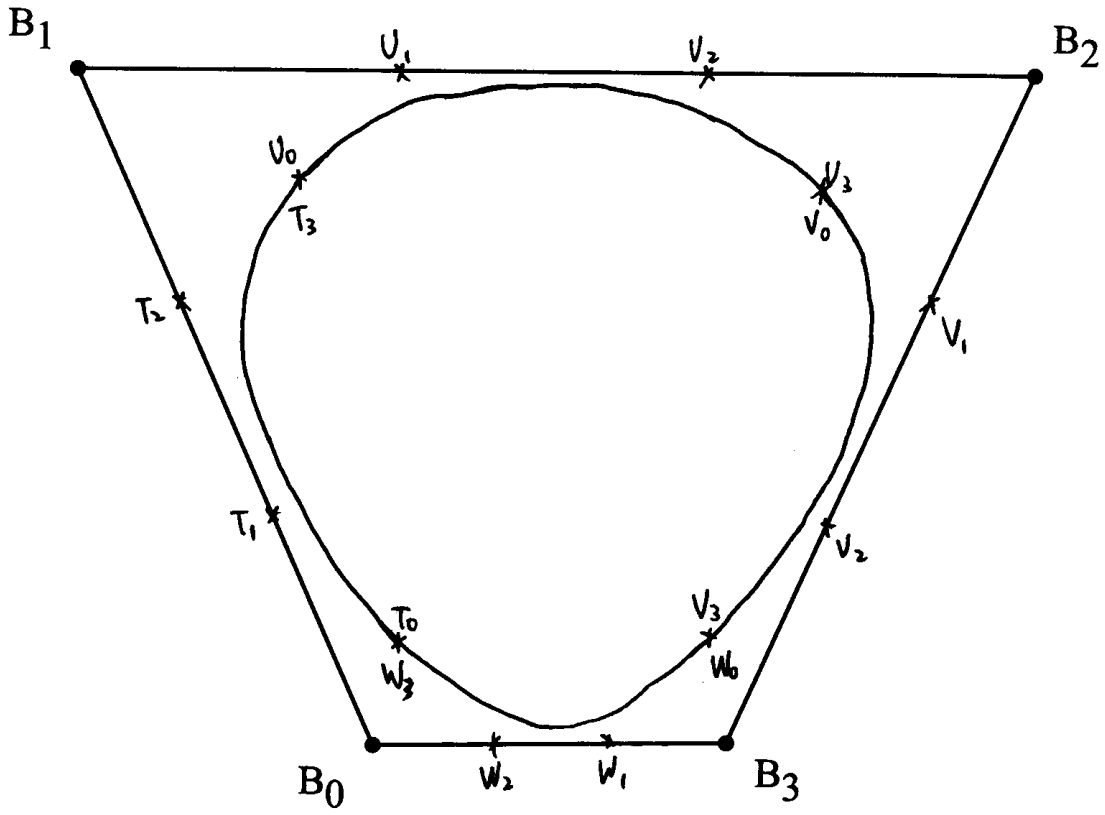
Problem 8. Constructing Splines

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. 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).

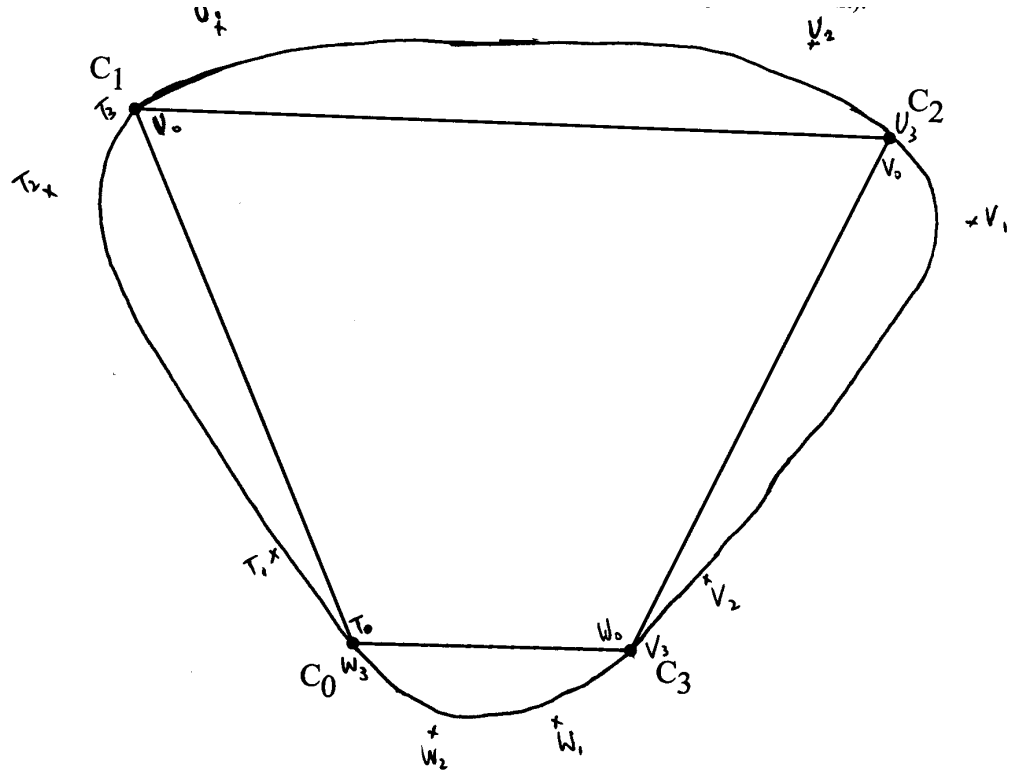


b. If you move one control point in your sketch, will the whole curve change? Justify your answer.

In this case, yes. Since there are four Bezier curve segments in this B-spline, and that each de Boor point determine up to four segments. So the whole curve will change when one of the above four control points is changed.

Constructing Splines (cont'd)

c. 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).



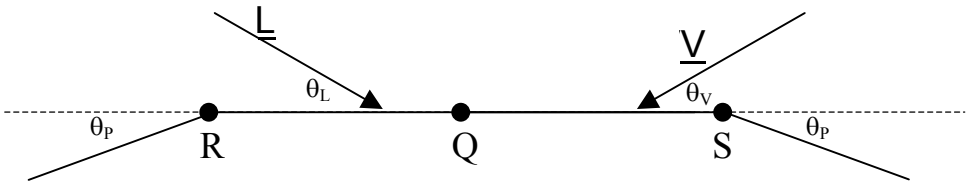
d. If you move one control point in your sketch, will the whole curve change? Justify your answer.

(Note the convex hull property is not necessarily satisfied by Catmull Rom curve in general. This is a counter example).

In this particular case, yes. Changing one point will render many, if not all of, the Bezier convex hulls be changed.

Problem 9. Polygon Shading

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 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 θ_V is equal everywhere.

e. 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:

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

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 $n_s = 100$.

N•L = cos(60) = 0.5
V•R = cos(0) = 1.0
$1.0^{ns} = 1.0$
$k_d * 0.5 + k_s * 1.0 = (1.1_R,1.5_G,1.1_B)$
$* I_L = (110_R,150_G,110_B)$

Polygon Shading (cont'd)

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

Yes, it does, for a number of reasons:
1. The term multiplied by k_s had sufficient magnitude to make it a significant contributor to overall intensity.
2. The viewing angle is identical to the reflection angle, so the specular highlight is guaranteed to be maximal at this point.
3. The overall intensity was greater than the intensity of the light source itself, which tells us that it could not all have come from just diffuse reflection, which at best reflects 100% of incident light.

g. 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.

Point R:
$\mathbf{N_R \bullet L} = \cos(50) = 0.643$
$\mathbf{V \bullet R_R} = \cos(20) = 0.940$
$0.940^{ns} = 0.002$
$k_d * 0.643 + k_s * 0.002 = (0.131_R, 0.645_G, 0.131_B)$
$* I_L = (13.1_R, 64.5_G, 13.1_B)$

Point S:
$\mathbf{N_R \bullet L} = \cos(70) = 0.342$
$\mathbf{V \bullet R_R} = \cos(20) = 0.940$
$0.940^{ns} = 0.002$
$k_d * 0.342 + k_s * 0.002 = (0.0686_R, 0.344_G, 0.0686_B)$
$* I_L = (6.86_R, 34.4_G, 6.86_B)$

Point Q:
$(k_R + k_S) / 2 = (9.98_R, 49.45, 9.98_B)$

h. 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.

It does not. Because the specular highlight did not fall on one of the edges, it was not part of the color interpolation which determined the color at Q. Unlike Phong interpolation, Gouraud interpolation has no way to know about specular highlights in the middle of a polygon because it does not calculate the normals at those intermediary points. It can also happen that a specular highlight will appear larger than it should, because it's right on a vertex and the highlight color gets averaged into all the neighboring polygons rather than being tightly focused as it should.

Polygon Shading (cont'd)

i. 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 **V•R** in the Phong lighting equation with **N•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:

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

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

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

The above equation is quite efficient to interpolate – rather than interpolating normal vectors, all we must do is interpolate the angle θ_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?

No, it does not produce a satisfactory approximation, and it does not produce specular highlights in the middle of polygons. It is approximately equivalent to Gouraud interpolation. The reason this method does not work is that the approximation it is based on is flawed to begin with, and that flaw carries forward to eliminate the desired benefit.

Problem 10. Ray Tracing

a. Our method of ray tracing involves tracing all rays from the eye. This is also known as "backward ray tracing." What are the major problems associated with this method?

One major problem is that the eye ray may not ultimately reach any light source in the scene.

b. If instead we decide to trace all rays from each light source, would we solve all of the above problems? What new problems arise by this method of "forward ray tracing"?

The light ray being traced may not finally reach the eye, the latter of which contribute to the shading of a pixel.

c. In class, we described how we could perform pixel anti-aliasing with jittered supersampling. Why is jittering better than casting rays with a purely random distribution? Given an example to illustrate your point.

One problem of random supersampling is that clustering may occur. The library functions available are in fact pseudo-random number generators, as it is a real difficult mathematical problem to generate numbers uniformly distributed in [0,1] while they should be random. Clustering results in negligence in some area, which is not desirable in ray tracing.

When we subdivide the square region formed by four pixels uniformly, and apply jittering to each subarea, we can guarantee that our random samples are distributed in the square. No particular region gets more or less attention in antialiasing.

d. Distributed ray tracing allows us to achieve more realistic results than simpler methods. For the following uses of distributed ray tracing, briefly describe how they are implemented. Indicate when only a few samples per pixel will suffice and also indicate when many samples would instead be necessary to mode the effect accurately (i.e. to avoid aliasing in other dimensions). List extreme cases where applicable. The first one is done as an example.

Pixel Anti-alising:

Randomly distribute a number of rays over the sub-areas of the pixel (jittering). Average the intensity contributions from these rays to determine the overall pixel intensity. Few samples would suffice for an area of uniform color. Many samples would be needed for a hard edge.

Motion blur:

Ramdomly distribute rays temporally as well as spatially. Before each ray is cast, objects are translated or rotated to their correct position for that frame. The rays are then averaged afterwards to give the actual value. Objects with the most motion will have the most blurring in the rendered image. Given objects of the same size, the faster the object moves, the more samples are needed.

Depth of Field:

Distributed ray tracing creates depth of field by placing an artificial lens in front of the view plane. Randomly distributed rays are used once again to simulate the blurring of depth of field. The first ray cast is not modified by the lens. It is assumed that the focal point of the lens is at a fixed distance along this ray. The rest of the rays sent out for the same pixel will be scattered about the surface of the lens. At the point of the lens they will be bent to pass through the focal point. Points in the scene that are close to the focal point of the lens will be in sharp focus. Points closer or further away will be blurred. Hence, the larger the difference in the depth range of the scene (i.e., the distance between the closest and the farthest object with respect to the viewer), the more rays are needed to simulate the depth of field.

Glossy reflections:

Glossy surfaces are generated in distributed ray tracing by randomly distributing rays reflected by a surface. Instead of casting a single ray out in the reflecting direction, a packet of rays are sent out around the reflecting direction. The actual value of reflectance can be found by taking the statistical mean of the values returned by each of these rays. The number of samples depend on surface property, i.e., whether the surface is perfectly specular (e.g. mirror, in which less samples are needed).

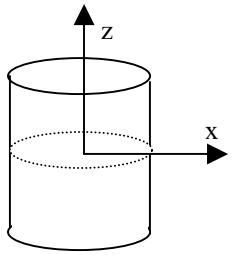
Problem 11. Procedural Solid Textures

a. One problem with the standard method of texture mapping is that textures appear to be stuck on objects in the same manner that wallpaper is put on a wall. In many cases, a surface doesn't look like it is really made up of the material that is texture mapped onto it, for example, consider a wood texture map. One method that is used to alleviate the "wood veneer" problem is to use a solid, or procedural, texture map.

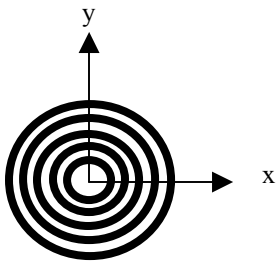
A procedural texture map answers the question: What is the color of the surface at this point?

Fill out the following function with very general pseudocode. Your code will return the answer to the question: What is the color of the wood at this point on the surface? Assume that you have values for the age of the wood, and the two colors for each type of ring in: age, darkRing, and lightRing. The Point passed into the function contains x, y, and z coordinates of the point on the surface in object space.

Here is an example of a horizontal slice of a cylinder with our wood shader applied to it:



Original Cylinder



Horizontal slice

Color wood (Point pt) {

Compute L = sqrt (pt.x*pt.x + pt.y*pt.y), which is the perpendicular distance to the z-axis.
Year = L*age / (width of a tree ring you defined);
return ((int)(Year) %2) ? darkRing : lightRing;

}

b. As the illustration above shows, this pattern is somewhat boring. It looks too perfect. How would you modify the code to introduce a less ‘sterile’ appearance, specifically, is there a way to introduce some `wiggles’ into the rings?

There are many ways to do this. One criterion is the year rings should still be approximately concentric, i.e., the year rings should not be intersecting each other. Therefore, we can add a random factor to L, make use of a deformable template, or use other functions. As long as the concentric requirement is satisfied (i.e., it still look like tree rings), it is fine.

c. Let’s say that during the seasons that yield light rings, the tree was absorbing silver from some strange atmospheric conditions. How would you modify the above function so that light colored rings were shinier than dark colored rings?

The specular coefficient can be increased for the light colored rings.

Problem 12. BSP Tree

Answer the following questions regarding BSP trees

a. Is it sort-first, sort-last, or both?

Sort first. Objects are sorted when the tree is built in the preprocessing step.

b. Is it image-precision, object-precision, or both?

Object precision in that objects are split and the tree constructed at the object level, but a painter's algorithm really determines visibility at the pixel level so you can argue it is both.

c. Is it image-order, object-order, or both?

Object order. The algorithm renders objects one after another. Visibility is not determined pixel by pixel.

d. Do all polygons need to be rendered?

Yes, in the basic algorithm objects are drawn from back to front, we can.t skip any polygons because we don't know if they will be occluded until everything is drawn.

e. Is it an online algorithm?

No, all the object data must be available for the tree to be constructed and rendering to begin.

f. Can it handle transparency?

Yes, it is easy to handle transparency with a painter's algorithm. Transparent objects just alpha blend with objects behind them which have already been drawn.

g. Can it handle refraction?

No, refractive objects don.t simply blend their color with the color further along the line of sight. They may potentially let you see around objects. Raytracing is the simplest way to handle refraction.

h. Can shading be done efficiently?

No, because all polygons must be rendered. (On the other hand, because whole polygons are rendered you can make use of efficient incremental rendering algorithms.

i. What are the memory requirements proportional to?

N, where N is the number of polygons, in the best case and 2N-1 in the worst case (every polygon split at every step in construction.)

j. What types of 3d graphics programs are well suited to BSP trees?

Walkthroughs where the viewpoint is changing but the scene is static, architectural visualization and first person games.