Assignment P3 : RAY TRACING

Computer Graphics 2 (15-463)

Due 11:59pm 11 April 1996

For this assignment you will extend your ray caster from assignment P2 with features to increase realism: recursive ray tracing with specular reflection and refracted transmission, shadows, distance-attenuated lights, automatic exposure control, and antialiasing.

Shading

The shading formula you should use is:

$$\mathbf{I} = \mathbf{C}_{mat} k_{dr} \Big[\mathbf{I}_a + \sum_i s_i \mathbf{I}_i \max\{(N \cdot L_i), 0\} / r_i^2 \Big] + k_{sr} \sum_i s_i \mathbf{I}_i \max\{(N \cdot H_i), 0\}^e / r_i^2 + k_{sr} \mathbf{I}_r + k_{st} \mathbf{I}_t$$

This is the same as in the previous assignment except for:

s_i is the shadow coefficient: 0 if light i is shadowed for this surface point
and 1 if light i is unshadowed

r_i is the distance of light i from the point being shaded
(this term included for $1/r^2$ attenuation of light)

$\mathbf{I}_r$ is the radiance of light in the specularly reflected direction (computed with recursive call)

$\mathbf{I}_t$ is the radiance of light in the specularly transmitted direction (computed with recursive call)

k_{st} is the coefficient of specular transmission

See Foley for explanations of these terms. In particular, I suggest you read sections 16.1.4, 16.5.2, and 16.12 before proceeding with the remainder of this handout.

To prevent infinite recursion on $\mathbf{I}_r$ and $\mathbf{I}_t$, only trace such rays to a maximum ray depth (specified at run time). Let's define depth=1 to mean ray casting with shadows, but no recursion.

Shadows. When testing for shadows, you can treat each object as opaque. You needn't do anything fancy to simulate partial shadowing of light through transparent objects.

Transmission with Refraction. When you compute the refracted ray direction (see Foley for formulas), you need to find the ratios of the indices of refraction of the material you're leaving and the material you're entering. When you're leaving air and entering a material (such as glass) with index of refraction "ir", then this ratio would be $1/\text{ir}$, but when leaving the material, entering air, the ratio would be the reciprocal of that. You'll need to do a test to determine if you're entering or leaving a solid. You can do this by comparing the incident ray direction D and the normal vector (exiting iff $N \cdot D > 0$). If there's total internal reflection, don't trace a transmitted (refracted) ray.

Normal Vectors. You'll probably need to be more careful with normal vectors on this assignment than in P2. In P2, your rays were always originating outside objects, so assuming the polygons in the scene file were all counterclockwise when viewed from the outside[1], the normals in your Scene data structure were all pointing toward the side of the surface you were viewing. In P3 we have transparent objects, so some rays will travel inside objects, and your code must be able to shade the inside surface of objects. To make this work right, after doing the entering/exiting test described above, make a copy of the normal vector, flip it if you're exiting the object, and use that for illumination calculations.

[1] we had some polygons backwards in room.sc for a while, but they're fixed now

Intersection Testing. Only the initial rays will have their origin at the eye point. The remaining rays (shadow rays and secondary rays for reflection and transmission) will originate at surface points. Say the ray equation is $X = P + tD$. If you don't take precautions, roundoff errors in intersection calculations can cause intersections to be detected at tiny positive values of t, with the surface from which the ray originates. These can cause speckles in your shading. There are two standard workarounds for this bug: instead of looking for the first intersection for $t > 0$, look for $t > \epsilon$, where $\epsilon = 10^{-4}$, say; or ignore intersections with the object from which the surface originates. Take your pick.

Antialiasing and Automatic Exposure Control

You should antialias by tracing multiple rays per pixel, adaptively. Since we're simulating distance attenuation of lights in this assignment, moving a light around could have large effects on the overall brightness of the scene, and could cause the final picture to come out overexposed or underexposed. To prevent that, we will use automatic exposure control, analogous to that found on smart cameras.

For antialiasing and automatic exposure control, implement something at least as good as the following: Make one pass over the image, generating one ray per pixel, saving RGB radiances as a vector of 3 floats or doubles. During this pass also compute the average of the maximum channel value of all pixels: avgmax = $\sum_{x,y} \max\{r_{x,y}, g_{x,y}, b_{x,y}\}/(\#\text{ pixels})$. This is a crude global measure of the brightness of the scene. Because of distance attenuation, this will not necessarily be near 0.5.

Then go back on a second pass, at each pixel testing the color there against the color of the four neighbors (above, below, left, and right) and supersample the center pixel 16 times (in a 4×4 grid) if any of the four neighbor's colors differs from the center color above some threshold. A good way to implement the threshold test is to compute $\Delta r^2 + \Delta g^2 + \Delta b^2$ where Δr, Δg, and Δb are the differences between red, green, and blue components, respectively, and if this sum of squares is greater than $(\text{avgmax} \times \text{thresh})^2$ then the colors are considered "too different". The parameter "thresh" should be some fraction between 0 and 1; perhaps .1 or so (bigger makes it faster, smaller makes the picture better).

You're free to use stochastic sampling (Foley page 788) or recursive adaptive sampling if you're interested (page 714).

On the second pass, you can write out pixels. If you multiply your floating point RGB radiance values by 128/avgmax then that should guarantee a reasonable range of pixel values (not too dark, not too bright), since it maps the average radiance to pixel value 128. In spite of this, some of your pixel values might overflow (> 255), so you should peg at 255 to prevent this.

Test Scenes

We've set up three test scenes in `pub/scene`:

table.sc A simple test of shadows and specular reflection rays. Same file as before, but now that you're simulating specular reflection, it will look snazzier.

glass.sc A careful test of specular transmission rays.

chex.sc A tough test of antialiasing.

See the `README` file in `pub/scene` for a more complete description of each scene. Also see reference pictures in `pub/pix/p3`, and the `README` file in that directory for an explanation of the pictures.

Programming Tips

In case you didn't solve assignment P2 completely, a good solution is being provided in `pub/src/p2/raycast`. You can build on that code for P3, if you like. The following modules are recommended for P3. This is one way to do things, but certainly not the only way.

`void trace_eye(double x, double y, Color *col)`
: Trace an "eye ray" (a ray leaving the eye) through screen space (x,y), and return the RGB radiance (color) of that ray in `col`. On the first pass, you probably want rays at $(ix + .5, iy + .5)$, where ix and iy are the column and row indices of the pixel. This routine simply generates a ray and calls `trace`.

`void trace(int depth, Ray *ray, Color *col)`
: Trace a ray, and return the color of light flowing backward along that ray in `col`. This ray is at depth `depth` in ray tree. Return black when `depth` gets to user-specified limit. This routine calls `intersect` and `shade`.

`Object *intersect(Ray *ray, double *t, Object *hier)`
: Determine the first valid intersection of the ray with objects in the hierarchy `hier`, and if any, return its t value in `*t`, and a pointer to the object hit. This routine recursively traverses the hierarchy to find the intersection point. It doesn't shade.

`void shade(int depth, Ray *ray, Object *obj, Point *p, Point *n, Color *col)`
: Shade the intersection point of `ray` with `obj`, which occurs at point `p` with normal vector `n`, where `depth` is the current depth in the ray tree. Return the color in `col`. Call `trace` at `depth+1` for specularly reflected and transmitted rays. Since `trace` calls `shade` and `shade` calls `trace`, they're both recursive.

What to Submit

Everything you submit should be in *yourdirectory*`/p3`. Submit the following:

1. Source code to your `raytrace` program.
2. A working executable of your `raytrace` program that we will be able to run (on either an SGI, Sun4, or LINUX).
3. TIFF picture files for `table.sc`, `glass.sc`, and `chex.sc`. Call them `table.tiff`, etc. Do them all with antialiasing. Set your antialiasing threshold so the pictures look good, but don't take excessive amounts of time (we'll leave that judgement up to you). Use a maximum ray depth of at least 5 on all pictures.
4. A nice picture of a tree (handmade, fractal, or whatever) of your own creation, rendered with your ray tracer. Turn in a scene file `tree.sc` and TIFF file `tree.tiff`. High scene complexity (large number of objects) is desirable here. Careful use of `push` and `pop` to build the hierarchical bounding volumes can help make this render quickly.
5. A text file, `README`, where you summarize briefly what you've done, what each of the files in your directory is, a description of the new scene you created and how you made it, and instructions on how we can run your `raytrace` to reproduce `table.tiff`. For each picture, tell us:
   - machine type computed on,
   - maximum ray depth,
   - threshold for antialiasing,
   - compute time (seconds of CPU time),

- # pixels,
- average # rays per pixel (including antialiasing rays, shadow rays, and rays at all depths in ray tree).

Extra credit will be given if you implement a really awesome tree, texture mapping, foggy materials, CSG, quadric primitives, blobs, or other cool new surface types. If you do something out of the ordinary, please explain it in your `README` file and comment your code.