The Writeup: MATLAB Raycaster
Jay Slater
12/12/08

Introduction

So. As my masterpiece image renders in the background, here I am starting on the writeup. There are plenty of interesting things to say, too, let me tell you. The first is echoed in the readme, but in case you don't I'm saying it here too. The Matlab package I have has every single package known to man installed with it. I'm pretty sure the raycaster is entirely self-contained, but if it's not, here's fair warning. Onward!

Implementation – General Design

The raytracer is built object-oriented, since it seemed to me that that'd be the simplest way of handling things. Environments contain everything involved in a given scene: a camera and as many light sources and spheres as your heart desires. The camera, light, and sphere classes are self-explanatory, and rt_utilities is a place where I dumped a bunch of useful static methods.

Implementation – Coordinates

This is kind of an oops on my part—it's not exactly LAB. Z is along the camera's original line of sight, yes, and Y is to the right, but X actually increases down instead of up. By the time I discovered the issue I'd already gotten rather well along in the project, so I ended up just leaving it as is.

Implementation – Camera

Instead of going with a massive array of points a la suggested in the assignment, I decided on a slightly different representation: the camera is described by a viewpoint and a direction, and the points on the retina are generated as necessary. Since, as far as I can tell, rotations against a constant frame of reference are commutative, it also vastly reduces the time required to do a rotation operation. The calculations do end up happening (once an image is requested), but since any number of rotations in two axes will always simplify to exactly two rotations, time savings in the long run can be considerable. Also, having read the assignment a little bit more closely, it strikes me as that this way of doing things is great for being able to do complex rotations in one go. Because the camera points are assumed to be at the origin until the very end of the find_camera_point method, there's no need to fiddle with rotations around an arbitrary point, or to move the entire camera matrix (which doesn't exist to begin with).

I should note that for most of last night and early morning, I thought I'd badly screwed up the code. This was partially true—I'd been adding the camera location to the camera points before rotating them. After I fixed that, though, I was still seeing distortion, and only just now do I realize that the distortion was due to field of view settings. Ooops.

Too, the camera has an exposure property—images can get kinda dark otherwise. It's a simple multiplier against the brightness of a given point.

Implementation – Objects

There's not much to say—the potential number of spheres is infinite, and since z-buffering is implemented, they can be arranged in any way. I didn't test whether or not they overlap cleanly, but I expect it could cause issues with lighting.

I did skimp a little on further object features, I'll note. Albedo functions are not implemented, nor surface roughness. On the other hand, I feel that a simple high-albedo Lambertian surface shows off the nice part of this project the best. I also feel that glossy is kind of ugly with multiple light sources—the one in the back looks a lot better than the ones in the front, I think (as you'll see in the examples further down).

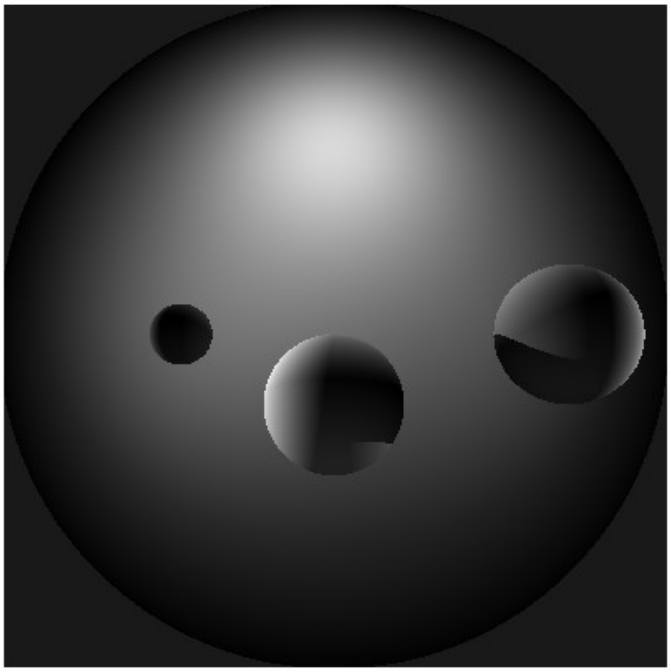
Implementation – Light and Shadow

Here's where I spent the lion's share of my time. The upshot—potentially infinite light sources at 3D points, radiating at a given brightness and decaying as light is wont to do. Also, all spheres self-shadow and cast shadows onto other spheres from all light sources.

Light incident on a point is a simple additive formula of light incident on said point from all light sources. Light strength is slightly more complex—the simple description is that I handle all light strengths without mucking about with checking the angle between normals by simply checking for anything in the way of the light—if there are any points of intersection on the segment between the point in question and the light source, then the light is 0.

I'm not sure if it was because I did so much work on this today, but getting the above method to work right was a serious headache. I ended up having to build in a fudge factor of .05 to the maximum distance to avoid artifacting on surfaces, which is why I worry about overlapping spheres. Oh well—something further to test.

Results



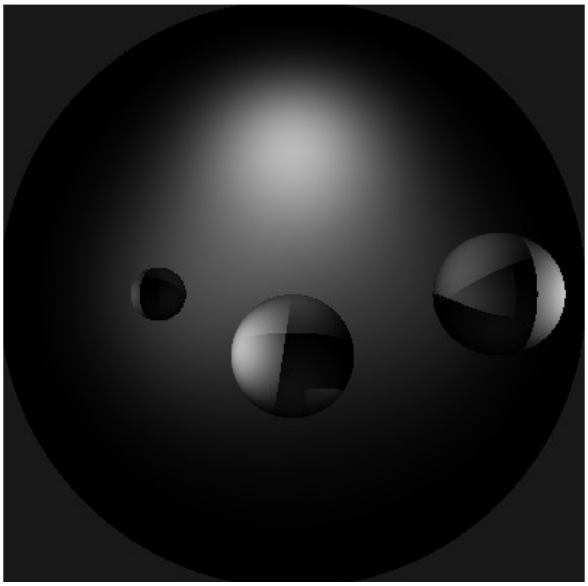
I thought the results were fairly impressive. To the side here is a picture of the arrangement of spheres and lights that the raycaster ships with (further down, at the end, is the : the large sphere in the background is radius 50 and 100 units away along the z-axis. The two medium-sized spheres are both 25 units away long the z-axis, with radii of 3. The small sphere is 19 units away along the z-axis and has a radius of 1.

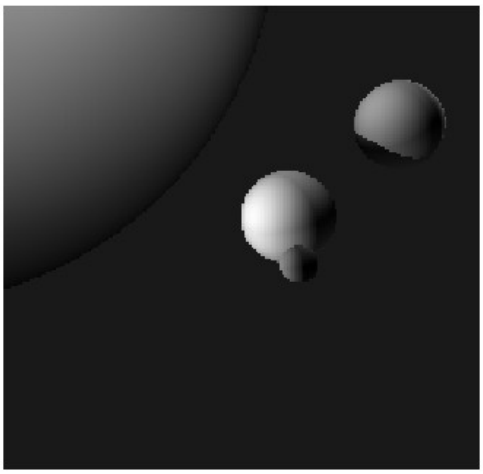
The light sources are arranged around the spheres thus: one is due left of the two medium spheres; another is due right. One is well below the two medium spheres, and one is above and in between the mediums and larges.

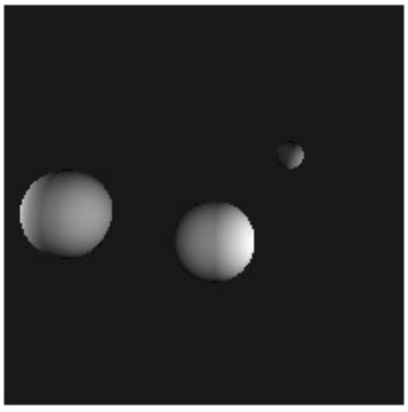
The first image has all of the spheres as full-on Lambertian surfaces, which nicely exhibits the shadowing I worked so hard on getting right. The camera settings were 400x400 resolution, with a focal length of 7, an exposure of 100, and a point spacing of .02. It took about 5-7 minutes to render. It's environment1 in the makethings script.

The second big image is glossy (I really can't help but feel I either did something wrong, or the hack we got in class simply isn't up to the task of so many different sources of light and shadow). The glossy image took significantly longer than the Lambertian one—I wonder if it has to do with the extra dot products?

The small images demonstrate some camera rotations (if you number them from top to bottom, there are some notes about them on the next page).







1. Took nearly half an hour to finish. Oooer. Environment 1 in the script.
2. A different perspective, lower resolution. Environment 3 in the script.
3. POV of the center light source looking down on the middle spheres. Environment 4. Notice how the light decay from this more distant source barely illuminates the spheres—the side light sources are much more pronounced.

A final note: I just spent about 15 minutes wondering why my images were suddenly so dark when I realized that I'd changed the code for light decay instead of glossiness.