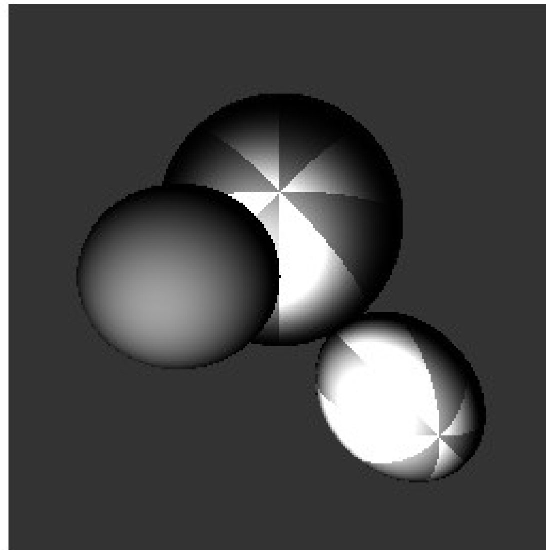
Sean Brennan
sbrennan@u.rochester.edu
CSC 173
Frames/MATLAB
12/17/09

MATLAB: Raycasting – with Spheres!

Introduction

This rather lengthy project involved writing a raycaster in MATLAB: that is, a program that displays 3D objects by casting "rays" out from a point in space. We treat this point, as well as a plane a fixed length away from it, as a camera, and if one of its rays collides with another object in space, the camera is able to see the object at the point of intersection. We store the brightness values of that point in a matrix, and we get our image.



Grayscale space watermelons.

Coordinates & Space

This project obviously required *a lot* of vectors. Fortunately, MATLAB is totally matrix-based, so representing three-dimensional vectors was easy: a 3x1 column matrix of the vector's components does the trick. The coordinate system I used is right-handed, though it's slightly different from the traditional LAB system. Rather than having X pointing up and having Y pointing right, I rotated the entire thing about Z by 90 degrees so that X pointed right, Y pointed down, and Z pointed away from the user. To me, this was more intuitive, as this is how most coordinate systems are defined in digital pictures,

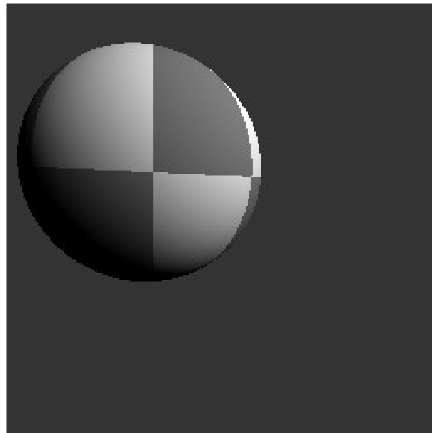
video games, and the like. It changed the way rotation matrices were defined, however.

The Camera

I used the basic pinhole camera model, which is a single focal point in space and a plane of pixels that acts as the image. The vector defined by the focal point and some point on the focal plane acts as a ray that we will project to "see" the objects out in space. This focal plane is always a fixed distance away from the focal point: this distance is called the *focal length* of the camera. The focal length can be manipulated to widen or tighten the image, and also do some other cool things.

This plane, and, as a consequence, the image, has a fixed *resolution*, though we can see more or less by messing with the *pixel ratio*, a measurement of how many pixels appear every arbitrary unit of distance. Finally, we come to the representation of the actual plane. CB recommended that we create a 2D matrix of 3D points for our focal planes, though I took a slightly different approach: I made the plane a 3xn matrix, where n is the number of pixels in the plane, and the sub-matrix (1:3, n) represents a pixel on the plane. While this doesn't seem incredibly intuitive at first, it made rotation a lot easier…I could just multiply the rotation matrix by the entire plane, rather than go to each pixel and rotate that, as the first and most obvious approach necessitates.

The camera has three basic operations: translation, rotation, and zooming. To translate the camera, we simply move the focal point to the new point, and add the resulting displacement to every pixel on the focal plane. Rotation is a bit tricky: first we have to move the camera back to the origin, rotate the entire plane by a rotation matrix given to us by the user, and then move back to the starting point. Zooming is as easy as pushing the focal plane further from the pinhole or pulling it closer. This reduces the angles between the rays cast and increases them, respectively, tightening and widening the image, respectively.



Rotating the camera 15° down, 15° right

The Objects of Choice: Spheres!

They are very exciting objects.

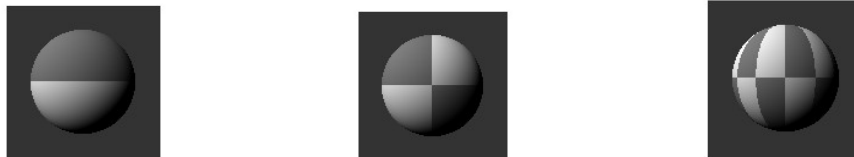
Spheres are very easy to implement in a raycasting programming because calculating the brightness of a surface in raycasting involves using the object's normal vector at a certain point. For spheres, the normal vector can be found simply by subtracting the center of the sphere from the point you're at, since every point on a sphere is equidistant from the center. No calculus or other forms of witchcraft are required.

Spheres are also handy because the formula for finding an intersection between a line and a sphere is significantly simpler than formulas for finding intersections between other objects and lines. (Not to say the sphere intersection equation isn't complex; it's ugly as all get out) The solution of the intersection formula reduces down to a special case of the quadratic equation, in which the term inside the radical gives us most of the information that we need…

$$\mathbf{b^2 + 4ac}$$

If this is negative, the line and sphere don't intersect at all; if it is zero, the line is tangent to the sphere, and there is one solution, **(-b/2a) * ray between focal and pixel**. If it is positive, then we have two solutions. But since our camera can only see one half of the sphere at a time, we choose the solution that's closest to the camera.

The sphere also has an *albedo* at every point, which is basically the "color" of the sphere at this point. I used CB's standard "beachball" sphere albedo, where there are a variable number of black and white stripes on the sphere. Giving it this appearance is easy: if the Y component of the sphere's normal vector is positive, then the sphere has a light appearance where Z is negative and a dark appearance where Z is positive; if the Y component of the sphere is negative, the opposite holds true. Changing the number of striped sections on the ball is as easy as multiplying the X-Z angle of the normal vector by the number of striped sections you want.



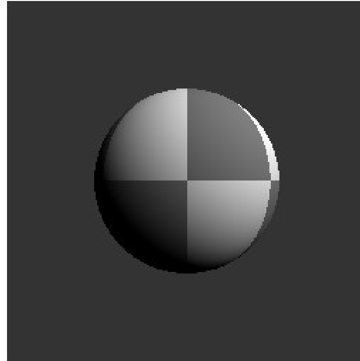
The wide variety of beach balls one can obtain with a simple albedo function is simply astounding.

Adding Light to Our 3D World

Of course, everything is meaningless without some form of light – we wouldn't be able to see anything without it! The easiest way to implement lighting in the raycasting program is "ambient light," a sort of omnipresent light that originates from nowhere and has only a direction. I also implemented light sources, which cast rays of light from finite points in space with variable intensity, but I'll get into that later.

There are a couple of ways for the sphere to "use" this light. The simplest way is

to make the sphere a Lambertian surface, which is a surface with a sort of uniform brightness that doesn't change when viewing the object from different perspectives. To find the Lambertian brightness for a certain point on the sphere, we need three values: the unit normal vector of the sphere at that point, the unit vector pointing in the direction of the light source, and the *albedo* of the sphere at that point, which is essentially a color value. With these, the brightness is simply the albedo multiplied by the dot product of the two unit vectors. Basically, the sphere gets darker the less the light is reflected and the more it is deflected off the sphere, until the light doesn't touch it at all and everything appears black.



A simple Lambertian beach ball.

Lambertian brightness is boring, though, so we also introduced specular reflection to liven things up a bit. This adds a bit of "mirrorness," as CB puts it, to the spheres, and also makes the brightness a function of the camera's location as well as the sphere and the light's direction. We need three dot products to compute this value…

I = sphere normal vector . direction to light
E = sphere normal vector . direction to focal point of camera
G = direction to focal point of camera . direction to light

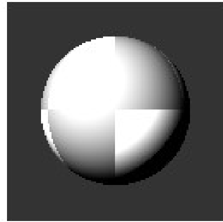
'.' here meaning the dot product of two vectors. We use these dot products to get our "mirrorness" value…

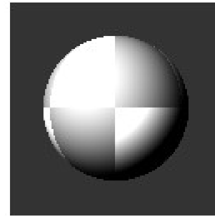
$$C = 2(I + E) - G$$

We can then combine everything together into this equation to get our final brightness:

$$sC^n + (1 - s)I$$

Here we get two additional factors. "s" represents the shininess or dullness of the surface: we can see that, by setting s to 0, we get plain old Lambertian brightness, while setting it closer to 1 will mean we have hardly any Lambertian factor at all. "n" is the degree of reflectance of the sphere: the higher it gets, the more receptive to light it will become. Multiply this specular brightness by the albedo, and we've got ourselves a nice-looking sphere again.

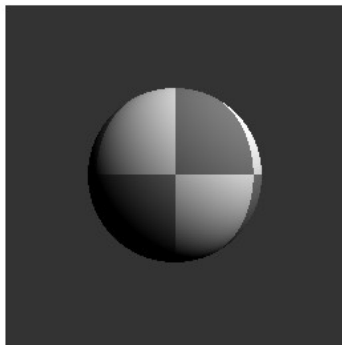


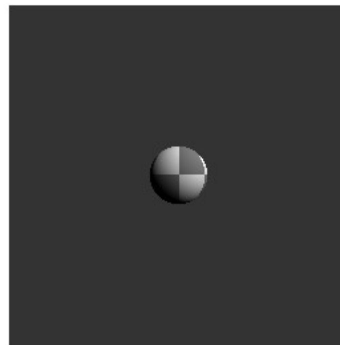


A sphere with an "s" of 0.5 and an "n" of 1.25, and another slightly duller sphere with an "s" of 0.25 and an "n" of 1.75.

Extra Credit Stuff

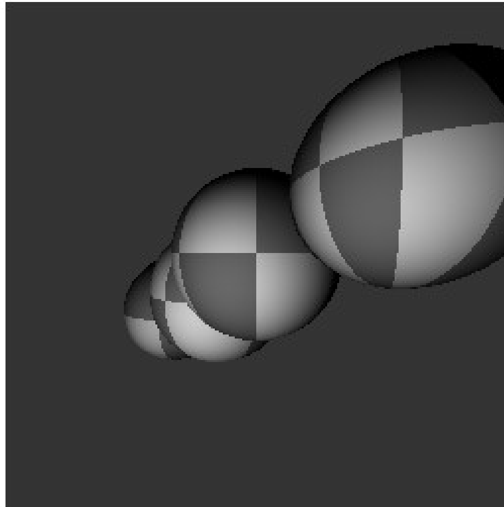
Zoom In, Zoom Out: As I explained earlier, my camera features the ability to zoom in and zoom out by a certain percentage. This is as simple as increasing the focal length to zoom in, and decreasing the focal length to zoom out. To change the focal length, I translated the camera back to the origin, changed the pixels' Z values, rotated it to the old camera's angles, and translated it back to the old camera's spot.





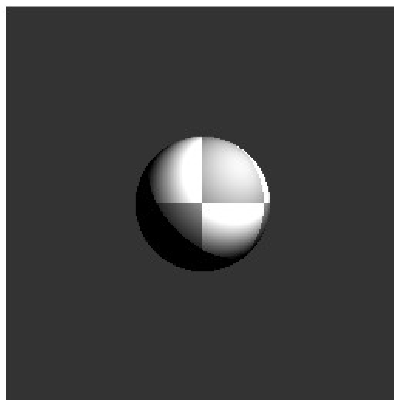
Focal length of 100, and zoomed out to a focal length of 50, respectively

Multiple Mutually Obscuring Spheres: My raycaster can support multiple spheres in the same space. Having more than one sphere adds a bit of complexity to drawing the image, since one sphere could be in front of another, from the camera's perspective. To work around this problem, I sorted the list of spheres by distance from the focal point, and then drew them from furthest to closest. If a ray intersects a closer sphere but there's already a point painted to that pixel, I just overwrote that pixel with the new sphere's brightness. If I had more time, I would have made spheres casting shadows on other spheres.



Spheres obscuring other spheres.

Finite Light Sources: My program features a single light source of variable intensity at a finite point in space. The major way that a finite light source differs from the ambient lighting at infinity is the way in which the light direction vector is computed: it changes as the point on the sphere changes, since the direction from one point on a sphere to the light source is always different from the direction from another point to the light source. I also threw “light intensity” into the mix, where the apparent brightness of the light at a given point is inversely proportional to the square of the distance from the point on the sphere to the point of light. I would have added functionality allowing for multiple finite light sources, but the rendering time for my images is ridiculous enough with only one!



A demonstration of a close light source, as well as some silly-looking self shadowing.