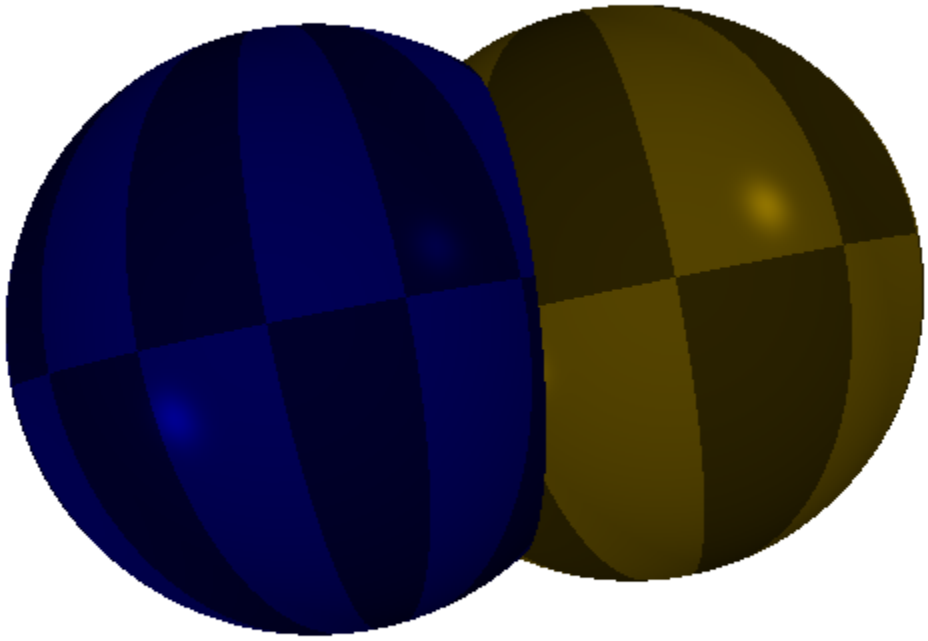


[Fig 1: Two colored mutually occluding spheres rendered under two white lights.]

1. The Camera and Simple Rendering.

The first step to establishing a solid ray-casting engine is to figure out the camera model. This is fairly simple: my camera consists of a focal point (the origin) and a rectangular surface on the XY plane of width “fieldOfView” and distance “focalLength”, attributes that can be configured (along with anything else about the camera). This rectangle is composed of an N by M array of points where N is the X resolution and M is the Y resolution, both passed in as functions. For each of these points, a ray is projected from the origin through the point into space, such that $F_x(z)=x/z$ and $F_y(z)=y/z$. Thus, a point on the ray is $P(z)=[z(x/f),z(y/f),z]$. Then, we simply iterate through every object in our universe, and see if the ray intersects the object, and figure out at what point that occurs. If it intersects multiple objects, we simply render only point with the closest non-negative Z.

We can find the intersection simply by noting that one occurs when the distance between the ray and the origin of a sphere equals the radius. In short,

```
sqrt(dx^2+dy^2+dz^2)-r=0
```

```
let A = sphere's origin: x
let B = sphere's origin: y
let C = sphere's origin: z
let x = ray's slope x (x/f)
let y = ray's slope y (y/f)
let z = z (the variable we're looking for)
let ^ = ^2 (inconsistently...)

(zx-A)^+(zy-B)^+(z-C)^                    = r^
z^x^-2Azx+A^+z^y^-2Bzy+B^+z^-2Cz+C^       = r^
z^x^+z^y^+z^-2Azx-2Bzy-2Cz+A^+B^+C^       = r^
z^x^+z^y^+z^+z(-2Ax-2By-2C)+A^+B^+C^      = r^
(x^+y^+1)z^+(-2Ax-2By-2C)z+(A^+B^+C^-r^)  = 0

QUADRATIC FORMULA!
(-b +/- (b^2-4ac)^.5)/2a
a = (x^+y^+1)
b = (-2Ax-2By-2C)
c = (A^+B^+C^-r^)

(+2Ax+2By+2C ± ((2Ax+2By+2C)^2-4(x^+y^+1)(A^+B^+C^-r^))^.5) / (2(x^+y^+1))
```

So it's pretty easy to get a value of z, and turn that into a point. But what if there is no intersection? We would get an imaginary number for the $(b^2-4ac)^{.5}$, so no problem: Matlab can handle imaginary numbers, but doing so is slow, so we'll just speed it up by culling all problems where this is the case. Furthermore, we can cut computation time by only taking the closer root (the $(-b-(b^2-4ac)^{.5})/2a$ case). If the z we get from this is negative, we're either inside the sphere or behind it: in either case, we don't want to render it, so we can drop all cases with a negative z.

Once we establish the z (if it occurs), we can get the point of intersection as mentioned before by simply remembering that `P(z)=[z(x/f),z(y/f),z]`. We can compute the surface normal of the sphere given this point and the origin, and the direction of light, and so on, using the sphere's attributes to get the ambient, Lambertian, and specular components of our light. In our model, ambient components (object-based) are identical to emissive components, so that type of light is not necessary to implement. Determining the shading is a simple process:

First, we look up our lighting rules. Each sphere is a struct that has fields for ambient, specular, and Lambertian light components per color channel (more on color channels later). Furthermore, each sphere has an albedo (also per color channel), which is simply a scalar value applied to our light function. Ambient light is simple: this component is always there and always at a strength of 1. Lambertian reflectance is zero or the dot product of the surface normal and the unit vector toward the light source (whichever is greater), and specular reflectance is calculated in a multi-step process:

1. Let “thetaC” be the dot product of the unit vector toward the camera and the reflectance of the unit vector of the incoming light by the normal vector.
2. Let b be the maximum of 0 and thetaC.
3. Let the final specular reflectance be b raised to the “specularconstant” - a variable defined in the setup of the camera.

Once we combine these values, weighted according to our sphere's attributes, we have a brightness for the pixel and move on to the next one. Once we've done every pixel, we pass the image off to “postprocesser.m”, which converts from our coordinate system to Matlab's: these coordinate systems are identical except that our X axis points up for positive numbers, and down for negative, while Matlab's i axis points up for negative numbers and down for positive numbers. After a simple vertical flip, we can generate pictures like figure 2.

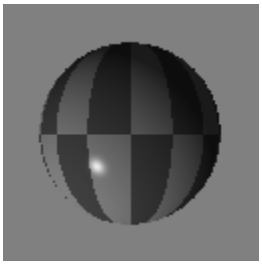


Figure 2

Figure 2 is a sphere painted with high and low albedo stripes similar to the ones pictured in the project description.

Furthermore, we can change the position of the sphere and the rendering adapts fine, as shown in figure 3. Here the sphere is shifted in the positive Z and positive Y directions by a small amount.


Figure 3

We can also, of course, rotate and track the camera, which is equivalent (in fact, implemented by) moving the entire universe around us in a rather geocentric fashion. The result of turning the camera “down” (pitch forward around Y) is shown by figure 4.

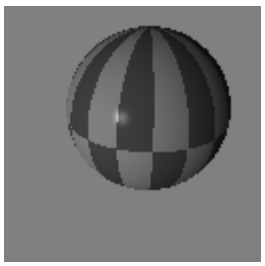


Figure 4

At this point, the bare-bones assignment specifications are completed, but what fun is it just sticking to gray-scale single spheres floating around?

2. Color Albedo Values

Color is very easy to introduce: we simply fall back on the traditional RGB color model and render three brightnesses: one per color channel. Our channels are naturally red, green, and blue. The first step to implementing decent color is to allow different albedo values for different color channels. Doing this, we can simulate a sphere, for example, with the albedo pattern on the red channel scaled by 1, green scaled by .4, blue scaled by 0 (meaning, no blue). This results in an orange-red sphere as shown in Figure 5. Although there is no figure showing it, the sphere shading components (Lambertian / specular / ambient) are also split across channels, allowing, for example, a turquoise matte sphere with an amber gloss.


Figure 5

3. Colored Lights

The next step in implementing full color is to allow the light to be different colors. This is fairly simple once we've implemented the color channels. We simply, instead of multiplying by the overall strength of the light when we determine the brightness of a point on the sphere, multiply by the color channel component of the light. So, a RGB light with strengths [1,1,1] is a white light, [1,0,0] is a red light, [1,.8,0] is a golden light, and so on. Figure 6, which demonstrates the light/object interaction, is the same sphere rendered in Figure 5, but with a light that has no green component, turning orange into red.


Figure 6

Figure 7 then shows what occurs if we turn the red component of the light off as well. As our sphere absorbs all blue light, and blue light is the only light around here, our sphere effectively becomes a blackbody.

Figure 7

4. Multiple Spheres, Mutual Occlusion

Even a single colored sphere is rather boring, and non-intersecting spheres, while fun, aren't terribly more so. Mutually-occluding spheres, however, can keep a small child entertained for hours. Our algorithm as mentioned in part one already accounts for multiple objects and mutual intersection: because we take the closest point on a ray-by-ray basis, we only care about the local behavior of the spheres: and thereby if they intersect each other elsewhere, we'll render that as appropriate but don't have to worry about it on the currently-enumerated ray.

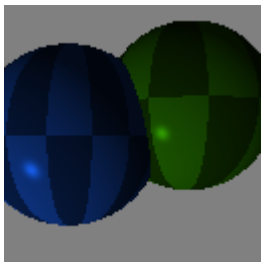


Figure 8

Also, because our spheres are handled as structures that have attributes, we're not bound to any global properties between these: it's not hard at all to have different colored spheres or spheres with different specular strengths. Figure 8 demonstrates mutual occlusion between two spheres, one blue, one yellow, rendered in a turquoise light (such that yellow becomes green due to the lack of red light).

5. Multiple Lights

Of course, since we iterate through objects during rendering, it's not much more difficult to iterate through all our lights when we're determining the brightness of a pixel. This does introduce a new concept that previously we've taken for granted: the dynamic range.

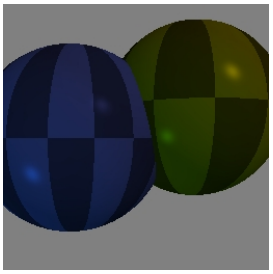


Figure 9

Simply put, what happens if our lights are so bright that our brightness rises above what we can express with our digital image? A human retina adjusts and limits the light in our system, so this program attempts (to some degree) to do the same. The idea is simply to multiply every brightness by a scalar to bring down the maximum possible brightness (2, if each light is 0 to 1) to our highest expressible value. More about dynamic range below.

Figure 9 are the spheres of Figure 8 with an additional white light. You will notice, due to the dynamic range damping, the turquoise appears less intense.

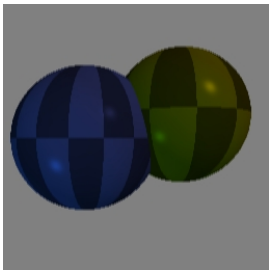


Figure 10

6. Post-processing Tricks

The post-processor flips our image vertically, but it doesn't need to be limited to just that. We can use it to implement any function that we perform on post-rendering, such as blur, linear/quadratic bloom, and so forth. Figure 10 is the spheres of Figure 9 (with the camera zoomed out) with a simple vertical 1-pixel blur applied.

7. Fine-Tuning the Dynamic Range

By fine-tuning the dynamic range we can come up with some better-looking pictures. There's no technical reason our lights have to range from 0 to 1 in intensity: Figure 11 is similar to figure 9 but with the white light at an intensity of 1.6. While this has the potential to result in some cases where we hit the top of our dynamic range, in this case we're pretty safe, as the two lights are coming from different directions and the sphere's specular percentages are so high. Of course, this has its limits: if we turn the light intensity up to 20, for example, we get Figure 12, which is usually not desirable. Figure 13 is a version of Figure 11 with even a slightly outside-our-range white light, though this is difficult to notice without close examination.

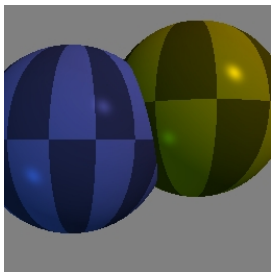


Figure 11

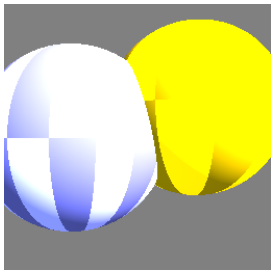


Figure 12

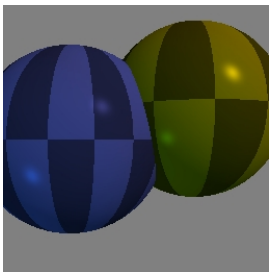
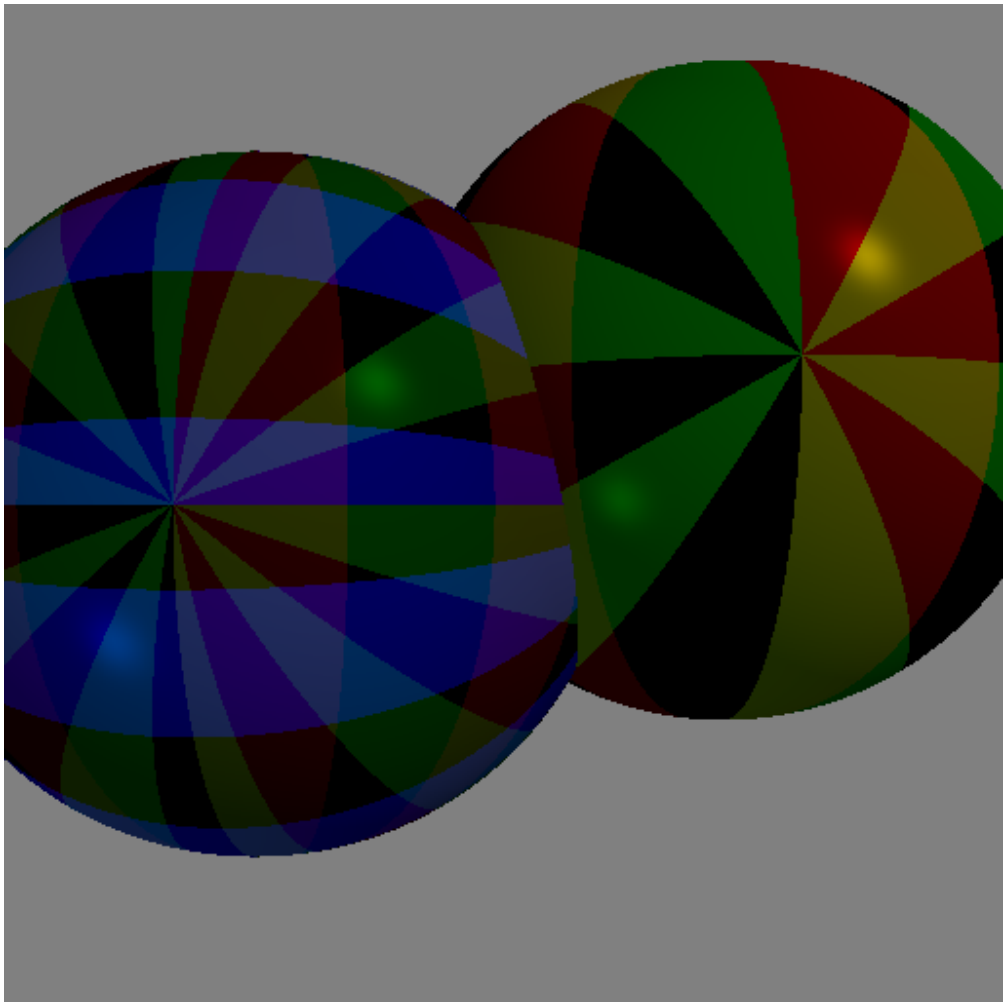


Figure 13



(figure 14, demonstrating that our RGB albedo values don't need to have the same function)