First-Hit Ray Tracer Report

By Alex Duffaut

2/12/2024

1. Scene

The scene I designed for this assignment consists only of simple objects. The objects include two spheres, one tetrahedron, and one infinite plane. All of these were implemented using the *Object* class, which has an *intersect* method to determine if any general object has been intersected, as well as contains different data fields that will be important to describe the intersection with the object. Those data fields include the material of the object, the normal of the hit, and the intersection point. The returned object that would store this information was called a *HitRecord*. Each ray that was shot off would generate a *HitRecord* even if it didn't intersect any objects. The default parametric value I used to represent infinity (which means no intersection occurred) was 100000. If a *HitRecord* returned a value of 100000, no meaningful intersections would happen.

The *Scene* class itself holds objects for the *Camera*, *Objects*, and *Lights*. Using other abstractions within those classes, those three objects describe the entire raytracing scene that I have drawn up.

The spheres are represented by a center and a radius squared. The figures for the two spheres I used are given here:

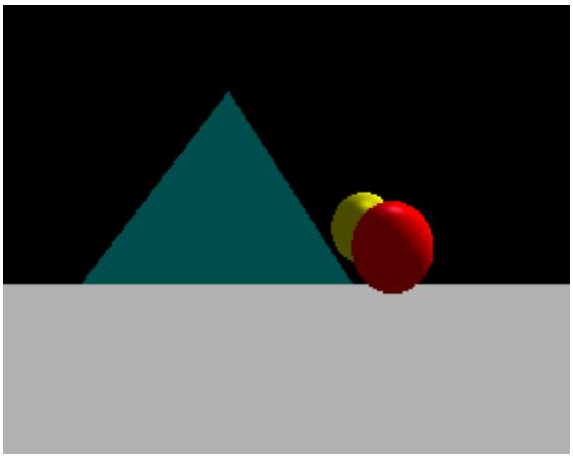
	Center	Radius2	Color
Sphere 1	(2, 0.4, -6)	0.25	Red
Sphere 2	(3, 1.1, -11)	0.5	Yellow

The vertices used by the triangles to describe the tetrahedron are given here (the tetrahedron is cyan):

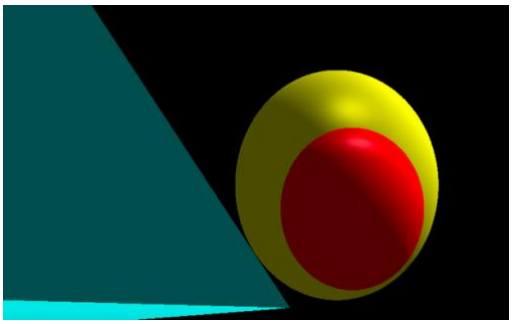
	Vertices	Triangles
V0	(-2, 0, -8)	Bottom (v0, v1, v2)
V1	(2, 0, -8)	Front (v0, v1, v3)
V2	(0, 0, -10)	Left (v0, v2, v3)

V3	(0, 3, -9)	Right (v1, v2, v3)

Finally, the infinite plane is the plane at *z = -1* and it is colored gray. All objects rest on top of the plane. Here is an image using a perspective camera of what the scene looks like from the origin facing down the Z-axis next to an image of what the orthographic camera looks like from the same angle:



Perspective



Orthographic

2. Camera

Two types of camera were used in my assignment. The first one I implemented was the *PerspectiveCamera* (even though I later found out that Orthographic was easier). The other camera that's part of my scene is an *OrthographicCamera*. I didn't separate them into different classes, rather I calculated the rays to shoot from the camera based off the Boolean *perspective* which told the program whether or not to use a perspective camera. Both cameras calculate their direction to the scene by using the *LookAt* vector method, which can be calculated in my program by passing in a *To* vector, and a *From* vector. Given the *To* and *From,* a user can angle the camera in any way they'd like and view the scene from all different angles. The *LookAt* vector is calculated from those two values using the *cameraToWorld* 4x4 matrix that transforms the camera space to world space. This matrix is calculated using the *Forward, Up,* and *Right* vectors that can be found using linear algebra from the *To* and *From* vectors.

2.1 Orthographic Camera

All the rays that come from the Orthographic Camera have the same direction, *From - To.* They all have different origins too. The calculation of the origins comes from drawing

a screen in the world space and shooting a ray from each pixel on that screen in our direction.

Using a double for loop to loop through values of 'i' and 'j' less than width and height respectively, this is how to calculate the x and y values on the screen in world space.

```
float x = (2 * (j + 0.5) / (float)width - 1) * aspectRatio;
float y = (2 * (i + 0.5) / (float)height - 1);
```

2.2 Perspective Camera

The rays that come from the Perspective Camera all have different directions, but the same origin. The directions are calculated by drawing a screen in world space and shooting a ray from our scene origin (which is always 1 single unit behind the screen) through the pixel in the screen. This creates the 3D effect of looking at the scene that looks more natural than the Orthographic Camera.

The same equation as above is used to calculate the screen in world space, except this time, the x and y values are used to calculate direction instead of origin.

2.3 Demo Scene

The scene starts by looking through a perspective camera. I did this because the Orthographic Camera zooms in way too far and doesn't view the whole scene. By pressing the "p" key after starting the program, you can render the scene from the perspective of the other camera.

	From	To
Perspective	(0, 0, 0)	(2, 1, -9)
Orthographic	(0, 0, 0)	(2, 1, -9)

3. Shading/Colors

Every *Object* in the scene contains information about the *Material* that makes up the object. This *Material* defines the shading that will occur upon a ray intersecting an object. The method for doing this was by holding three different types of coloring: *ambientColor (ca), surfaceColor(cd), and specularColor(cs).* There are also three factors to hold the intensities for the three colors: *ambientIntensity (ka), surfaceIntensity (kd), and specularIntensity (ks).* I also have the classes hold a *mirrorIntensity (km)* but I was unable to get the mirror/glazed property of objects to work in this assignment. I have left light intensity out of all my

calculations since the values are all 1.0 for every light source The table below lists the properties of the objects in my scene:

	Ca	Cd	Cs	Ka	Kd	Ks	km
Sphere1	(255, 0, 0)	(255, 0, 0)	(255, 255, 255)	0.3	0.6	0.3	0.0
Sphere2	(255, 255, 0)	(255, 255, 0)	(255, 255, 255)	0.3	0.6	0.3	0.0
Tetrahedron	(0, 255, 255)	(0, 255, 255)	(255, 255, 255)	0.3	0.6	0.3	0.0
Plane	(255, 255, 255)	(255, 255, 255)	(255, 255, 255)	0.3	0.4	0.3	0.8*

Notice that even though the plane has a *km* of 0.8, I was unable to get any glaze on the object.

The total lighting on an object comes from the following equation:

L_total = ka * L_a + kd * L_d + ks * L_s

These *k* values are not supposed to be totaled above 1 to prevent light intensity that would overflow pixels, but I was unable to see spectral lighting very clearly without increasing the spectral intensity. These were the settings that made my scene look the most realistic.

A table of the lighting in my scene (all had intensity 1.0):

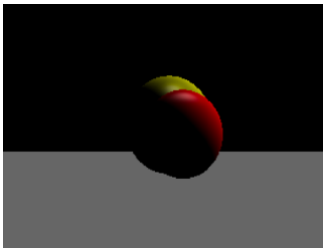
	Direction	Point Source
Ambient	N/A	N/A
Diffuse	(-1, -1, 0)	N/A
Specular	N/A	(4, 10, -8)

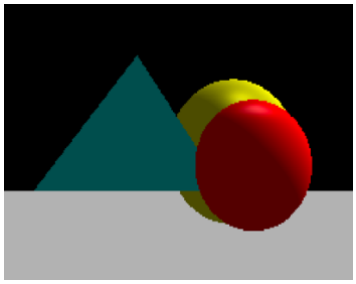
3.1 Ambient

Ambient lighting is very simply given by the ambient color times the ambient intensity. The L_a that would contribute to a color is simply *ca.* When I discuss the L values of the different colors, I am leaving out the intensity factors since those are seen in the formula above.

Here are pictures describing what the scene would look like with only ambient lighting, vs without, vs normal:


Ambient only


Diffuse + Specular

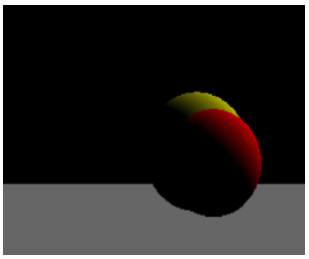

All Lights

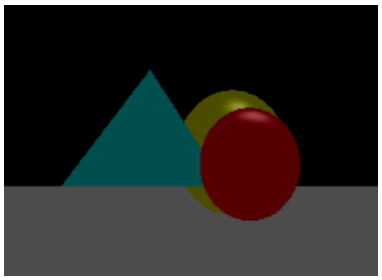
Note that for the diffuse and specular setting, the triangle disappears because the only face visible to the user at the time would be one that is not hit by a point or a distant light.

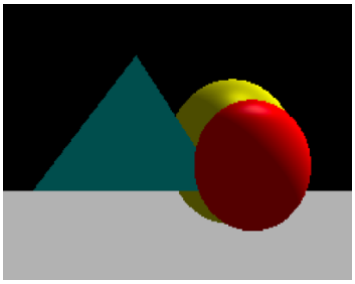
3.2 Diffuse

When calculating diffuse lighting, what matters is the normal of the object you are intersecting, as well as the direction of the diffuse lighting that is on the scene. For my scene, there was one single diffuse light with *direction = (-1, -1, 0)*. For calculating the diffuse, we will use a vectors *dir* and *normal*, as well as of course taking into account the diffuse color of the object we're intersecting. The formula looks like this:

L_d = cd * max(0, dir.dot(normal))


Diffuse only


Ambient + Spectral


All Lights

3.3 Spectral

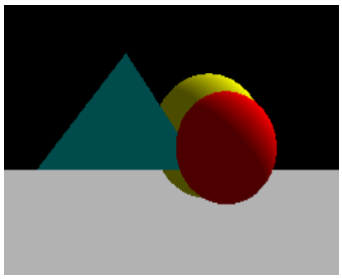
Since specular lighting comes from a point, we need to effectively shoot a light ray from that point, and see where it intersects the objects in our picture. The formula to do this requires a direction of our viewing ray, *dir,* our viewing *origin*, the point of intersection with the object, *phit*, the reflection of the light ray off of our object, *reflection,* the source point for our light, *point,* and the normal of the object at the point of intersection, *normal.* Using these, we can calculate the spectral lighting on an object using the following equation:

Vector X = (point – phit).normalize();

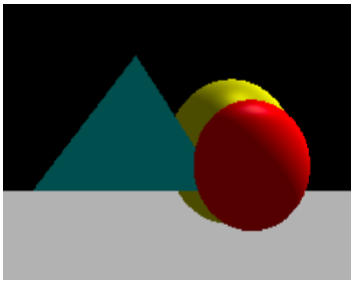
Reflection = 2 * (normal.dot(X)) * normal – X;

L_s = cs * max(0, reflection.dot(-dir);

Spectral only



Ambient + Diffuse



All Lights

4. Glazed Surfaces

I was not successful in completely implementing glazed surfaces. The idea I had was to add a recursive call to my *Object.intersect()* function that would shoot a new ray out from the point of intersection in the reflected direction of the initial ray, and that would test intersections for other nearby objects that would draw themselves on the mirror. This method timed out on me, because every time I would try to test it out, the program would never startup because of how many calculations it was trying to do.

The formula I used to calculate the new reflected direction for a vector with oncoming direction, *X,* and a vector normal, *normal,* was as follows:

Reflection = 2 * (normal.dot(X)) * normal – X;

5. Movies

I used *ffmpeg* and a GitHub gist from https://gist.github.com/maluoi/ade07688e741ab188841223b8ffeed22 to write my scenes to TGA image files, and then compile them into a short animated scene. I produced two total movies, animation1.mp4 and animation2.mp4, which are different perspectives on the same scene. Both of my animations targeted 24 frames per second.

5.1 Animation 1

The first animation is a simple circular scan that starts at the point (0, 0, 0) and spins around the point (2, 1, -9) on the XZ plane (without moving the y-values and going up and down). This allowed the movie to change the camera *From* vector, but keep a consistent *To* vector so that the objects in the scene didn't fly out of scene as soon as the animation started, and instead, you can view them from different angles. The formula I used to spin the camera around included a value *frame*, the current frame that the scene would be on, a value *time*, the total time it would take to spin around the y-axis, and a value *fps*, the target frames per second of my animation. Then, the *From* vector looked something like this to produce the animation:

From.x = 10 * sin(2*PI*frame / (time * fps))

From.y = 2

From.z = 10 * cos(2*PI*frame / (time * fps))

5.2 Animation 2

The second animation aimed to get a longer and closer look at the objects, and then zoom away from them to show the different lighting effects from vastly different angles. This was done by changing the following formula slightly, to look something like this:

From.x = 5 * sin(2*PI*frame / (time * fps) + THETA)

From.y = 2

From.z = 5 * cos(2*PI*frame / (time * fps) + THETA)

Where in this case, THETA was an angle of -120 degrees so that we can see the backsides of the objects before circling around to the front. Besides these changes, animation 2 had all the other same settings as animation 1