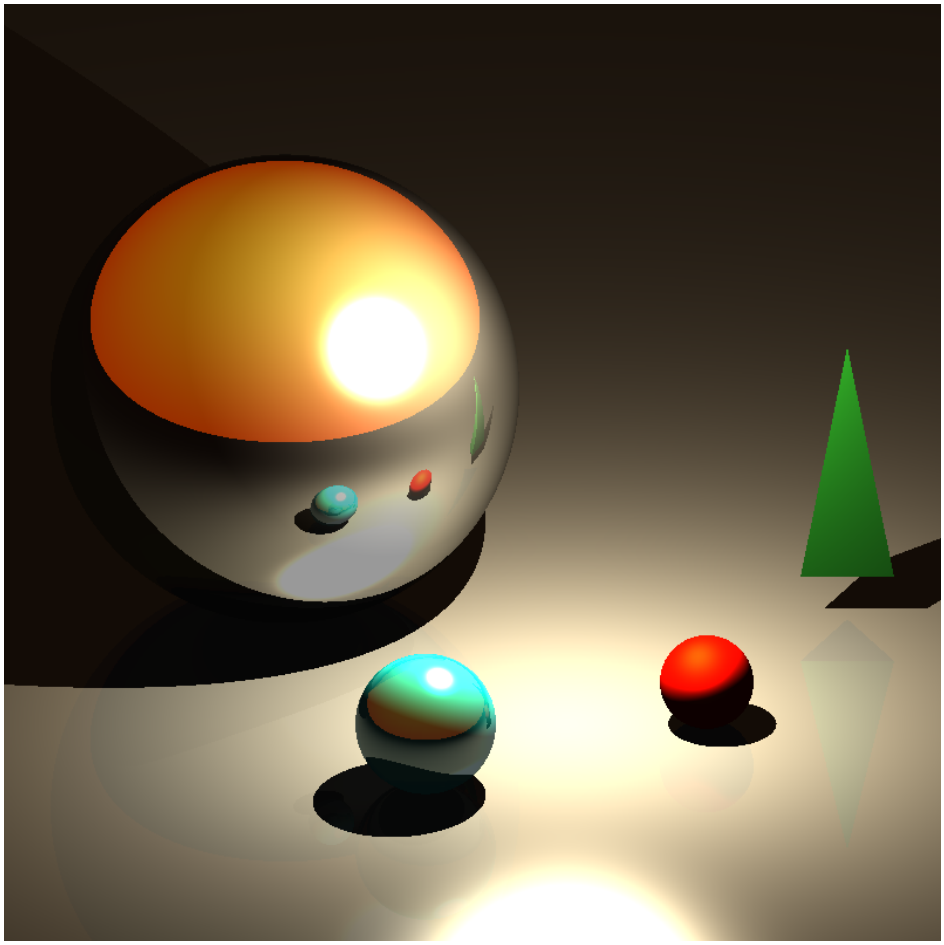
Computer Graphics Assignment 1: First-hit Ray Tracer

Ty Beller

February 2024



1 Introduction

For this assignment we are tasked with building a simple first-hit ray tracer. In the project as well as my report I drew heavily from the sample report provided.

2 Rays

The most basic type in this project. A ray has an *origin* and a *direction*. The *Ray* class contains the *at* function, which returns the point vector at a given t along the path of the ray

3 Material

The *Material* class defines the properties of each of the *Surfaces*. Each material has an independent color and strength property for *ambient*, *diffuse*, and *specular* light, allowing for iridescent materials. The material class also states the *phongExp*, if the object is *glazed*, and its *reflectivity*.

4 Surfaces

For the sake of this project I implemented three types of surfaces: *triangles*, *spheres*, and *planes*. All surfaces have a *material* and can calculate the intersection between the surface and a given *Ray* along the ray, and the surface normal vector given a point.

4.1 Triangles

4.1.1 Normal

The surface normal of a triangle where v_0, v_1, and v_2 are the triangle's vertices is shown below

$$\vec{v_1} - \vec{v_0} \times \vec{v2} - \vec{v0}$$

4.1.2 Intersection

For this project I used the Möller–Trumbore ray-triangle intersection algorithm to find the point t along the ray in which the ray intersects the triangle if at all.

4.2 Sphere

4.2.1 Normal

The surface normal of a sphere is simply the point on the sphere minus the center of the sphere.

$$\|\vec{p} - \vec{c}\|$$

4.2.2 Intersection

I used the analytic solution for ray-sphere intersection. We first find the vector $\vec{oc}$ from the center of the sphere to the ray's origin. Then we calculate the coefficients for our quadratic equation with

$$a = \vec{rayDirection} \cdot \vec{rayDirection}$$

$$b = 2 \times (\vec{oc} \cdot \vec{rayDirection})$$

$$c = \vec{oc} \cdot \vec{oc} - radius^2$$

After this we calculate the discriminant for the quadratic. If its less than 0 then we return -1 if its greater than or equal to 0 then we calculate and return the smaller root of the quadratic equation

4.3 Plane

4.3.1 Normal

The surface normal of a plane is given when the plane is instantiated therefore we simply return the stored normal value.

4.4 Intersection

As in the Möller–Trumbore implementation we calculate the intersection by the dot product between the surface normal and the ray direction. Once we know that it intersects we calculate t

$$t = \frac{planeDistanceAlongNormal - \vec{normal} \cdot \vec{rayOrigin}}{\vec{normal} \cdot \vec{rayDirection}}$$

5 Camera

There are two camera types in the project: *OrthographicCamera* and *PerspectiveCamera*. All cameras have a *width*, *height*, *viewPoint* vector which corresponds to the camera's position, *lookAt* vector which is where the camera faces, and an *up* vector. You may switch between the cameras by pressing the P button

From these we generate

$$\vec{e} = \vec{viewPoint}$$

$$\vec{w} = \|-\vec{lookAt}\|$$

$$\vec{u} = \|\vec{up} \times \vec{w}\|$$

$$\vec{v} = \|\vec{w} \times \vec{u}\|$$

Then we calculate $aspectRatio = \frac{width}{height}$ and multiply by 10 to get the *halfWidth* and just take 10 as the *halfHeight* to scale the viewing window to 10 units tall regardless of the resolution chosen

After this we can calculate the *pixelWidth* and *pixelHeight* to be double the corresponding half value divided by either the width or height.

We then use this to calulate the position of each pixel at position $[i][j]$ in our space as such

$$x = -halfWidth + pixelWidth \times (i + \frac{1}{2})$$

$$y = halfHeight - pixelHeight \times (j + \frac{1}{2})$$

5.1 Orthographic Camera

The orthographic camera has each ray having the same direction, but different origins corresponding to each pixel on the screen.

5.1.1 Ray Direction

All ray directions are simply $-w$

5.1.2 Ray Origin

For each ray, we calculate the position of each pixel as

$$\vec{origin} = \vec{viewPoint} + \vec{u}y + \vec{v}x$$

5.2 Perspective Camera

The Perspective camera instead has every ray keep the same origin, but shoot at the viewing pixel at a different direction for each pixel, making the window expand with distance creating perspective. As such the perspective camera requires another input, the *projectionDistance*, which is the distance from the origin of the rays to the viewing plane.

5.2.1 Ray Direction

The ray direction is calculated as

$$\| - projectionDistance \times \vec{w} + \vec{u}y + \vec{v}x\|$$

5.3 Ray Origin

All rays share the same origin at the *viewPoint*

6 Lights

There are three types of lights implemented in this project: *ambientLight*, *directionalLight*, and *spotLight*. While a scene can have any number of directional or spot lights, there is only one ambient light source per scene.

6.1 Ambient Light

Ambient light is treated differently than directional and spotlight and is not a child of the base *Light* class. It takes a *color* and *intensity* and has getters for these values. As ambient light does not depend on the viewing ray and there are no light rays, there is no math to be done.

6.2 Directional Light

Directional light, as stated, is a subclass of the *Light* class which has a *color*, *direction*, and *intensity*. All of these values are constant regardless of the point at which the light is calculated at (barring shadows but that math is handled elsewhere) therefore the class simply has getters to retrieve these values.

6.3 Spot Light

The spot light is the most complex light implemented in this project. It takes in two extra values: the light *position* and *maxAngle*. The *direction* and *intensity* of a light ray are entirely dependent on the position of the point on which we are calculating it.

We calculate the *direction* at a given *point* as

$$\|\vec{pos_{light}} - \vec{pos_{point}}\|$$

From this we can calculate the *angle* of the light from the centerline of the spotlight as

$$angle = \arccos\left(lightRayDirection \cdot lightCenterDirection\right)$$

We calculate the *intensity* at a given *point* as

$$intensity = \begin{cases} \frac{intensityCoefficent}{distance^2 * (1 - \frac{angle}{maxAngle}} & angle \leq maxAngle \\ 0 & otherwise \end{cases}$$

7 Scene Class

The *Scene* class holds all of the surfaces and lights and renders the scene. It has a light vector, a surface vector, an ambient light, and an output image. The most important function of the *Scene* class is the *renderImage* function which takes in a camera as input and renders the scene from it, saving the output to the image variable.

8 Shading

Shading is performed within the *Scene*'s *trace* function, which takes a *Ray* as input and returns the color vector for that vision ray. Throughout our shading we keep a running sum of the color starting off as a zero vector $(0.0f, 0.0f, 0.0f,)$.

8.1 Finding closest surface

First, we have to see, what the ray hits if anything at all. We iterate through every *Surface* in the *Scene* and run the *getIntersection* function with our vision ray. We keep a count of the lowest t value, meaning it was hit closest, and corresponding *Surface*. For the hit point on this surface we then iterate through every light source in the scene to calculate the shading.

If no surfaces are hit we set the color to change with the vision ray's z component, changing the color from red to yellow to create a sunset effect for the background.

8.2 Ambient Light Shading

With i representing intensity, c representing color, LA representing light ambient, and MA representing material ambient.

$$\vec{color} = \vec{color} + i_{LA} \times \vec{c_{LA}} \times i_{MA} \times \vec{c_{MA}}$$

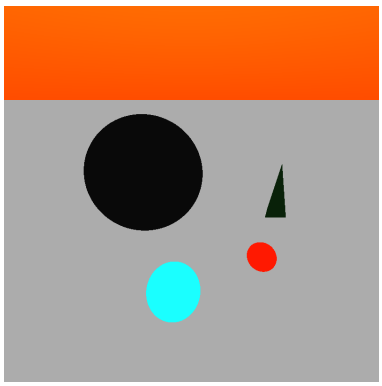


Figure 1: Scene with only ambient light (saturated for clarity)

8.3 Shadows

We calculate shadows independently for each non-ambient light source. At the hit point we create a ray that is in the reverse direction of the light source. Then we check every surface in the scene to see if it hits any of them. If it hits another surface then the point at the view ray is in a shadow for that light source and we do not calculate the diffuse or specular shading for it.

8.4 Diffuse Shading

Diffuse shading emulates a matte material appearance as it simulates the light diffusing off of the surface on reflection. For diffuse shading I implemented the Lambertian shading model. Lambertian shading is view independent, meaning that only the surface normal and light position impact the resultant color.

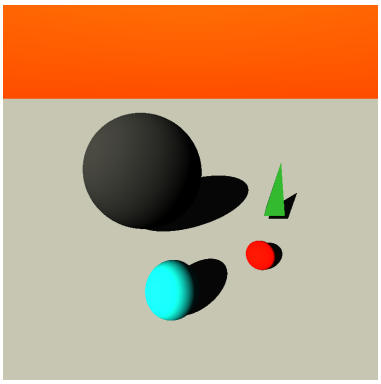


Figure 2: Diffuse, shadows, and directional lighting

First we want to check that the surface normal is facing outwards, which we do by taking the dot product of the vision ray direction and the normal. We have the vision ray going towards the hit point here (we reverse it later for the diffuse and specular math), so if this dot product is greater than 0, we reverse the normal direction.

To calculate the diffuse light on a point we calculate

$$diffuse = \vec{n_{surface}} \cdot \vec{lightDirection}$$

Then, we apply the scalar to the colors and add it to our output color.

$$\vec{color} = \vec{color} + diffuse \times i_L \times \vec{c_L} \times i_{MD} \times \vec{c_{MD}}$$

8.5 Specular Shading

Specular shading emulates the shininess and reflectivity of plastic and metal like materials. For specular shading I implemented the Blinn-Phong shading model. Blinn-Phong is not view independent, it requires the viewing ray to calculate the shading aswell as the surface normal and light ray.

Blinn-Phong requires the calculation of the *half vector* v_H, a vector which is half way between the viewing vector and the light vector.

$$specular = \max\left(0, \vec{normal} \cdot \vec{v_H}\right)^{phongExp}$$

Again, we take this value and apply it to our color

$$\vec{color} = \vec{color} + specular \times i_L \times \vec{c_L} \times i_{MS} \times \vec{c_{MS}}$$

8.6 Glazing (Recursive Ray Tracing)

Glazing is how we emulate a reflective surface. If a surface is glazed we find the reflection ray, which is the view ray reflected on the surface normal, and run the ray tracing algorithm on that as well to add to our color. This occurs regardless of whether or not our material has a light source hitting it.

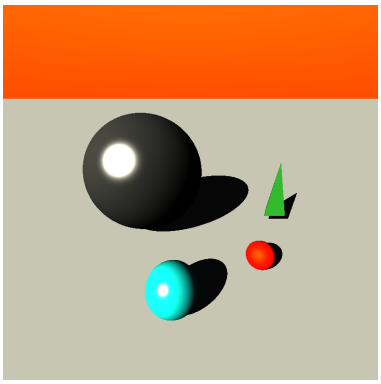


Figure 3: Specular, diffuse, shadows, and directional lighting

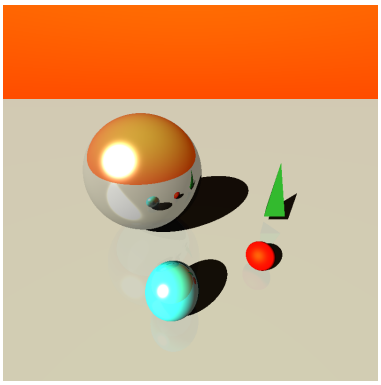


Figure 4: Glazing, specular, diffuse, shadows, and directionl lighting

9 Scene

As seen in the pictures, the scene contains 3 spheres, a plane, and a tetrahedron made from 4 triangles.

The materials used in the scene are detailed below. The indivudual RGB values for ambient, diffuse, and specular light are identical for all of these materials and as such have been combined.

Name	R	G	B	Ambient	Diffuse	Specular	phongExp	Glazed	Reflectivity
Plane Mat	130	130	130	0.1	0.5	10.5	100	true	0.05
Red Mat	255	19	1	0.1	0.5	1.5	10	false	N/A
Teal Mat	20	255	255	0.1	0.3	10	100	true	0.3
Green Mat	70	255	70	0.01	0.21	5.5	79	false	N/A
Metal Mat	70	70	70	0.01	0.21	5.5	79	true	0.6

Table 1: Material Properties

The parameters of each sphere are as in Table 2. The parameters of the plane are as in Table 3 with the X, Y, and Z values refering to the normal vector. The parameters of each triangle in the tetrahedron are in Table 4. For the scene, there is an ambient light, a spot light, and a directional light, although in the renderings only one of the spot and directional light are on at a time. Their paramaters are in Table 5

The scene contains 2 cameras, the orthographic and perspective cameras with parameters in figure Table 6. You can switch between the two cameras, which will trigger a re-render, by pressing *P*.

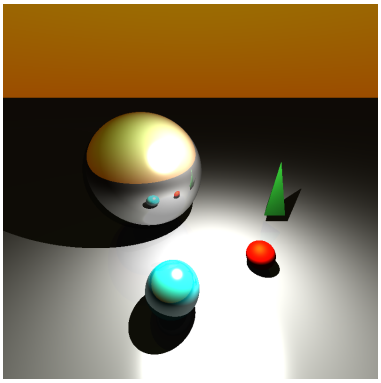


Figure 5: Spot light

Radius	X	Y	Z	Material
1	-3	0	0	Red Mat
1.5	-3	-3	0.5	Teal Mat
5	6	6	4	Metal Mat

Table 2: Sphere properties

Distance along Normal from Origin	X	Y	Z	Material
-1	0	0	1	Plane Mat

Table 3: Plane properties

X_1	Y_1	Z_1	X_2	Y_2	Z_2	X_3	Y_3	Z_3	Material
-5	5	0	-7	5	0	-6	7	0	Green Mat
-5	5	0	-7	5	0	-6	6	5	Green Mat
-5	5	0	-6	6	5	-6	7	0	Green Mat
-7	5	0	-6	6	5	-6	7	0	Green Mat

Table 4: Triangle properties

Name	R	G	B	dir_X	dir_Y	dir_Z	intensity	pos_X	pos_Y	pos_Z	maxAngle
Ambient	255	255	255	N/A	N/A	N/A	0.002	N/A	N/A	N/A	N/A
Directional	255	255	230	-50	50	-50	0.02	N/A	N/A	N/A	N/A
Point	255	255	255	0	0	-1	2.2	0	0	10	2π

Table 5: Light parameters

Name	Width	Height	VP_X	VP_Y	VP_Z	$lookAt_X$	$lookAt_X$	$lookAt_X$	up_X	up_X	up_X	projDist
Ortho	800	800	2	-10	10	0	2	-1	0	0	1	N/A
Persp	800	800	2	-10	10	0	2	-1	0	0	1	10

Table 6: Camera parameters

10 Movies

From the scene and two cameras, I created two movies. The code for generating the images used to compile the movies is commented out in the code to reduce load times. ffmpeg was used to combine the images

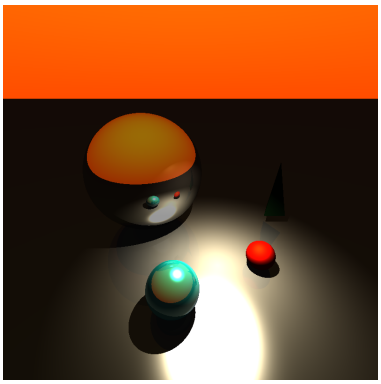


Figure 6: Spot light with $\frac{\pi}{4}$ maxAngle

generated with stb_image into the movies.

10.1 Movie 1

Link: Movie 1

Movie 1 uses the perspective camera and the spot light.

To move the camera I set the viewpoint as below with i representing the frame number.

$$viewPoint_X = -10 + 20 \times \frac{\sin(2\pi \times i)}{seconds \times fps}$$
$$viewPoint_Y = -10$$
$$viewPoint_Z = 16 + 15 \times \frac{\cos(2\pi \times i)}{seconds \times fps}$$

The lookAt vector is then set to the normalized negative of the viewPoint vector, making the camera always look directly at the origin. This creates a sweeping effect which has the camera swoop around the scene.

10.2 Movie 2

Link: Movie 2

Movie 2 uses an orthographic camera and directional lighting. The viewPoint is set as below.

$$viewPoint_X = -10 + 20 \times \frac{\sin(2\pi \times i)}{seconds \times fps}$$
$$viewPoint_Y = -40$$
$$viewPoint_Z = 26 + 15 \times \frac{\cos(2\pi \times i)}{seconds \times fps}$$

Here you can see that the Y value of the viewPoint has been scaled up from -10 to -40 which, as the perspective camera doesn't show depth perception, makes the swoop from the X and Z changes simply appear more subtle. The look at vector is again set to the normalized negative of the viewPoint vector.

11 Requirements

Built with latest versions of gcc, GLEW, and GLFW. Images and videos generated with stb_image and ffmpeg.

12 Build Instructions

Build with makefile by running "make" in the "/bin" directory

13 Useful Links

- https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm
- https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shape ray-sphere-intersection.html