First-Hit Ray Tracer Report
CAP 5705 – Assignment 1

Mohammad Anas, Audrey DeHoog

1 Objects

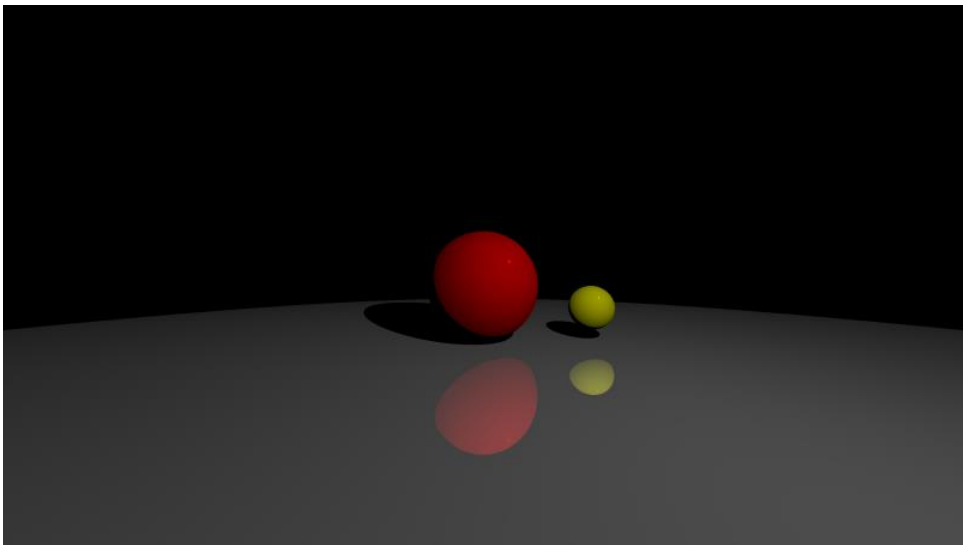
The scene we designed contains three spheres, one large sphere for the ground and two smaller spheres next to each other. The spheres are implemented using the *sphere* class (*sphere.h*) that contains a constructor and a *hit* method that calculates the sphere normal vectors.

The *sphere* class is a child of the *hittable* class (*hittable.h*). The *hittable.h* contains a *hit_record* class, a *hittable* class, and a *hittable_list* class. The *hit_record* class is used to store information about an object in the scene and for setting the face normal with the function *set_face_normal*. The *hittable_list* class contains a vector (*objects*) that contains all the objects within the scene. The class *hittable_list* also handles checking if a ray has hit an object in the scene.

The scene itself is set up in the *main()* of *main.cpp*, which creates the image, the world, and the camera information. The surfaces in the demo scene are created by creating a *hittable_world* object and adding surfaces to it with the constructor. A sphere is created by providing a *point3* containing the center of the sphere, the radius of the sphere, the color given by a *vec3* of RGB values, and a boolean value that determines if the glazed material is applied. The information on the three spheres created in the demo scene can be seen here:

	Center	**Radius**	**Color (R, G, B)**	**Glaze**
Sphere 1	(0, 0, -1)	0.5	(1, 0, 0) [red]	False
Sphere 2	(1, -0.2, -1)	0.2	(1, 1, 0) [yellow]	False
Sphere 3 (ground)	(0, -100.5, -1)	100	(0.5, 0.5, 0.5) [gray]	True

The two spheres exhibit only shading, while the ground sphere has a glazed surface. The sphere positions were chosen so that they rest on top of the ground sphere. Below are the outputs of the demo scene:



2 Camera

The camera is set up in *main.cpp* in the *main()* function. By default, the camera focal length is set to 1.0, the viewport height to 2.0, and the viewport width is calculated by multiplying the viewport height by image width/image height. The image aspect ratio, image width and height, and scale are also set up in the *main()* function.

There are two types of cameras implemented: orthographic and perspective. The Boolean *orthogonal* controls the toggle between the two camera types.

2.1 Viewing Rays

Ray-generation begins from the camera viewpoint and from basis vectors {u, v} where viewpoint vector {u} is across the horizontal and viewpoint vector {v} is down the vertical viewport edges. We then calculate the location of the upper left pixel in the image.

The *ray* class (*ray.h*) contains the constructor of the ray, stores the origin and direction of the ray, and handles the function representation of the ray.

$$\boldsymbol{R(t) = p + td}$$

Where ***R(t)*** is the ray, ***p*** is the origin point, and ***d*** is the direction of the ray.

The camera then constructs and dispatches rays into the world. The returned results of these rays are used to construct the rendered image.

To ensure that the image comes out clearer, we implemented antialiasing. This is done by getting a randomly sampled camera ray for each pixel. The count of random samples for each pixel is controlled by the variable samples_per_pixel .

2.2 Orthographic Camera

If the Boolean *orthogonal* is set to true, the viewing ray origin is set to the pixel sample value and the direction is set to a negative vec3 of (0, 0, 1).

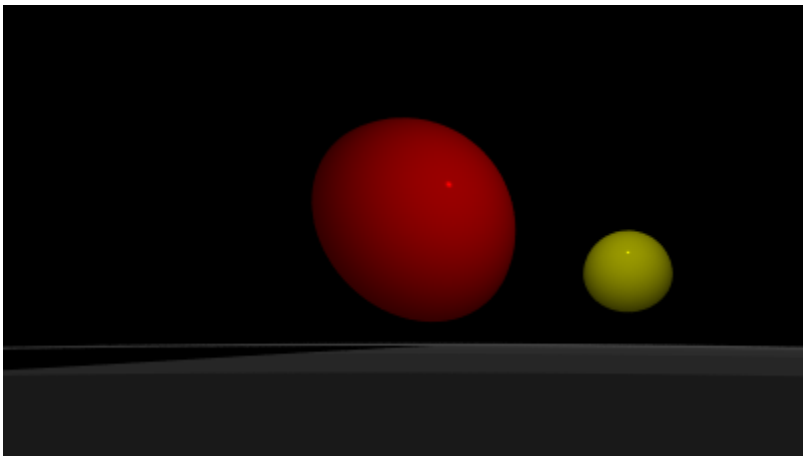
2.3 Perspective Camera

If the Boolean *orthogonal* is set to false, the viewing ray origin is set to the center of the camera and the direction is set to the ray direction.

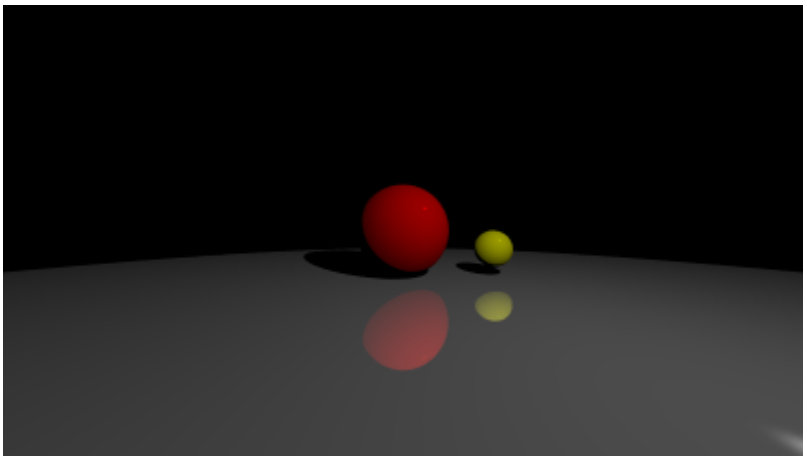
2.4 Demo Scene

The function *DisplayImage* in *main.cpp* opens a GLFW window that displays the output of the demo. The output is also saved into png files.

Demo with orthographic camera:



Demo with perspective camera:



3 Shading

Shading is handled in *main.cpp.* Once the ray is created, the ray, world, and depth values are then passed to the *ray_color* function that calculates the ambient shading $\boldsymbol{L_a}$, the diffuse shading $\boldsymbol{L_d}$, and the specular shading $\boldsymbol{L_s}$.

$$\boldsymbol{L} = \boldsymbol{L_a} + \boldsymbol{L_d} + \boldsymbol{L_s}$$

The function *ray_color* returns a color value that is added to the image data to represent the RGB values of the objects in the scene. The *color* class (*color.h*) is simply a representation of RGB values in the range [0, 255] that are stored in a *vec3*. When converting the *color* class value to integer values, the color

values are passed to the function *clamp* in the *interval* class (*interval.h*) To ensure that the color components of the result remain within the proper [0,1] bounds. The *interval* class contains helper functions that are used throughout the ray tracing process.

The constants for the shading calculations are

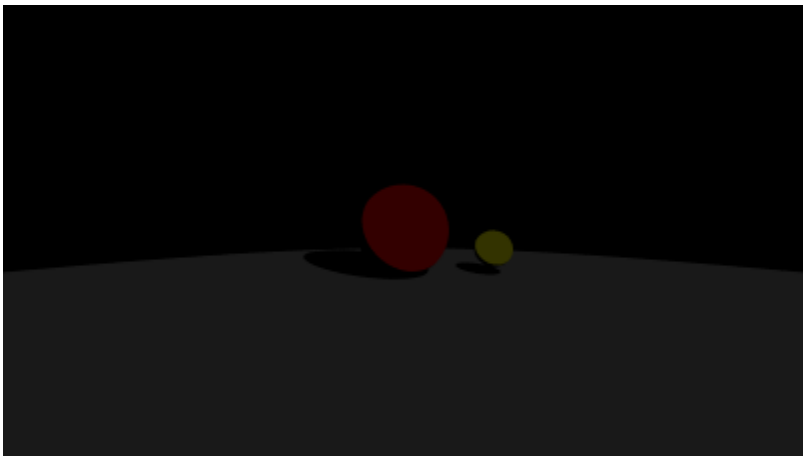
Constant Name	**Value**
AMBIENT_STRENGTH	0.2f
DIFFUSE_STRENGTH	0.3f
SPECULAR_STRENGTH	0.6f
MIRROR_STRENGTH	0.5f
SHININESS	10000.0f
MAX_DEPTH	50

3.1 Ambient

The ambient lighting of the surface is calculated by multiplying our AMBIENT_STRENGTH constant (represented by I_a) by the color of the object the ray interacts with (represented by k_a). The intensity of the ambient lighting is controlled by the constant.

$$L_a = k_a I_a$$

Below is an image of the demo with ambient shading only.



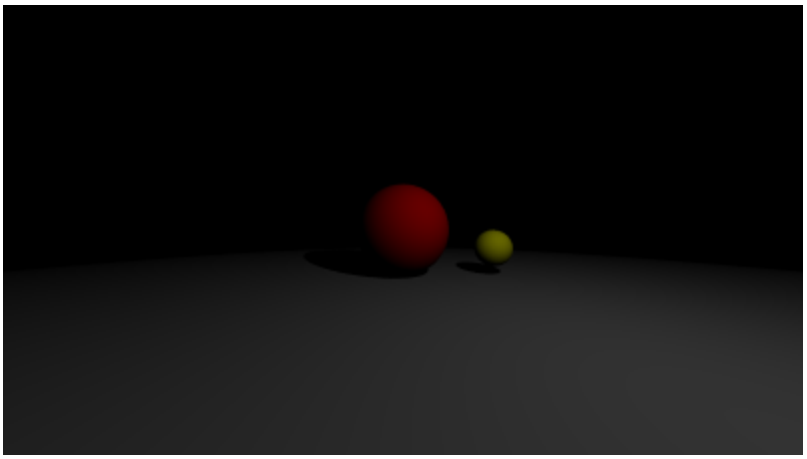
3.2 Diffuse

The diffuse lighting of the surface is calculated by multiplying our DIFFUSE_STRENGTH constant (represented by I_d), the color of the object the ray interacts with (represented by k_d), and the dot product of the light direction (represented by V_L) and the surface normal (represented by n).

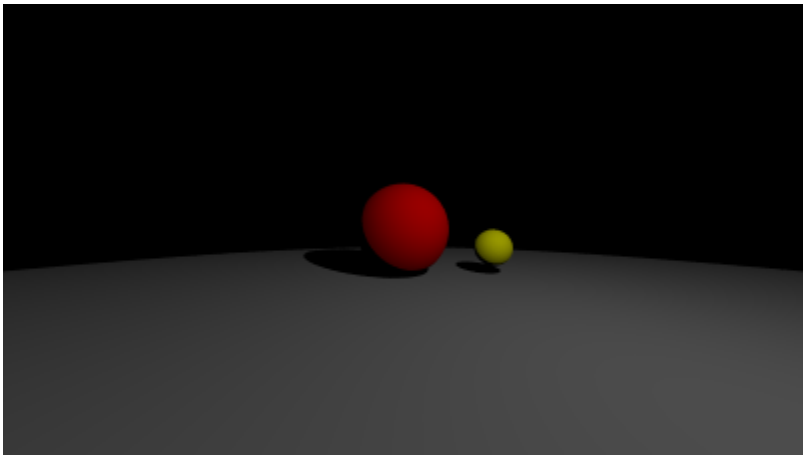
$$L_d = I_d k_d \max(0, n \bullet V_L)$$

The dot product and unit vector calculations are all handled by the *vec3* class (*vec3.h).*

Below is an image of the demo with diffuse shading only.



And now with both diffuse and ambient.



3.3 Specular

The specular lighting of the surface is calculated by multiplying our SPECULAR_STRENGTH constant (represented by I_d), the color of the object the ray interacts with (represented by k_d), and the shininess (represented by $max(0, n \bullet V_H)^p$). The intensity of the specular lighting is controlled by the constant. The shininess is calculated by taking the maximum between zero and the dot product of the normal to the surface (represented by n) by the direction of the light source (represented by V_L). This is then multiplied to the power of our SHININESS constant (represented by p).

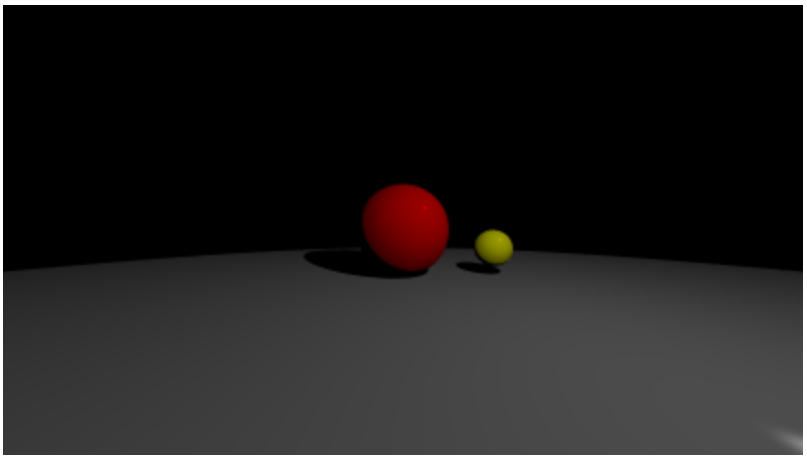
$$L_s = I_s k_s max(0, n \bullet V_H)^p$$

Rather than computing reflection directly, the function *reflect_half* computes the half vector between the light source's direction and the viewing ray's direction.

$$V_H = bisector(V_L, V_E)$$

The reflected ray, dot product, and unit vector calculations are all handled by the *vec3* class (*vec3.h).*

Below is an image of the demo with ambient, diffuse, and specular.



4 Glazed Surface

A glazed surface has mirror reflections of other objects. If an object in the scene has its glazed Boolean set to true, the *ray_color* function in *main.cpp* will add the glaze calculations to the returned ray color. The intensity of the reflectiveness is controlled by the constant MIRROR_STRENGTH.

The reflected ray is calculated by

$$\boldsymbol{r = d - 2(d \bullet n)n}$$

Where ***d*** is the direction of the viewing ray and ***n*** is the normal of the surface. This calculation is handled by the *reflect* function in the *vec3* class.

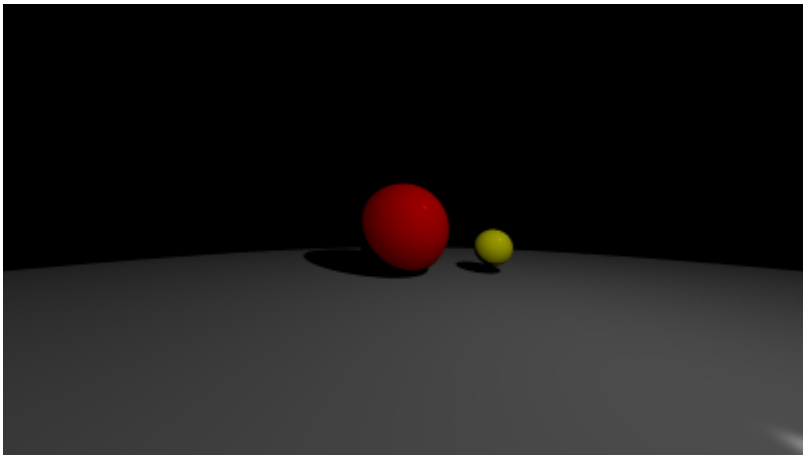
The glaze is then calculated by

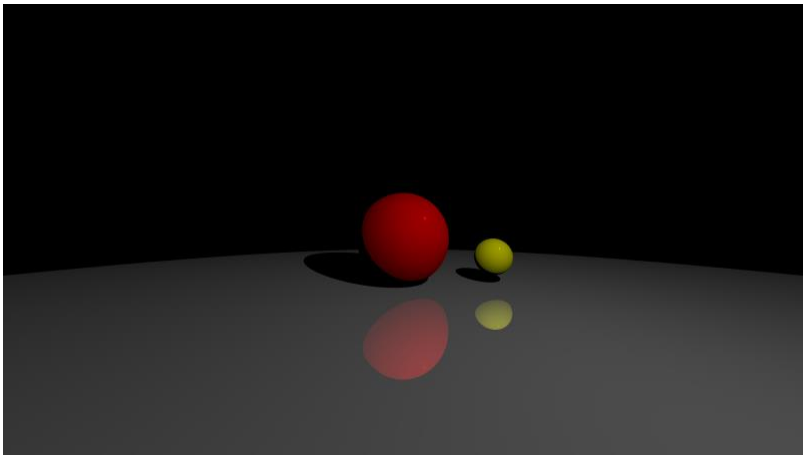
$$\boldsymbol{Color = c + k_m\, raycolor(p + sr, \epsilon, \infty)}$$

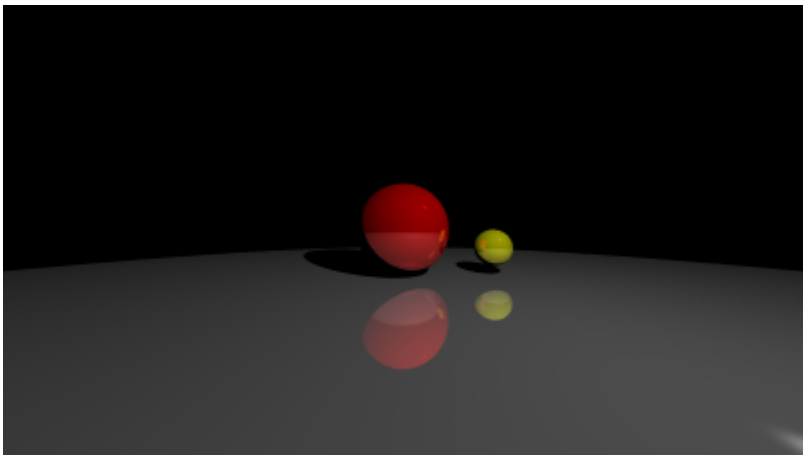
Where *ray_color* is recursively called with the reflected ray. This is done until the depth value (starting at zero) becomes greater than the maximum depth of the reflection (indicated by the MAX_DEPTH constant). Once the glaze recursion is complete, it is added to the ambient, diffusion, and specular color.

$$\boldsymbol{L = L_a + L_d + L_m}$$

Below are images of the demo without glaze, a glazed plane, and all objects glazed.







5 Animation

The animation is created by producing multiple images from the ray tracer and storing them in a "frames folder". The variables animation_duration, frames_per_second, and total_frames in *main()* control the time settings for the animation. The camera center is updated with each frame.

To convert the frames into a video, we created a python script called *video_create.py* that uses the OpenCV library. The Python file stores the PNG frames in an array and sorts them by name. The first frame of the video gives us the dimensions for the video. OpenCV's *VideoWriter* function combines all the frames into a video and saves it as "output_video.mp4".

6 Additional Information

The class *helpers* (*helpers.h*) was created to hold useful functions to use in the calculations. It only contains the function *random_double* that generates a random double based on a given max and min.

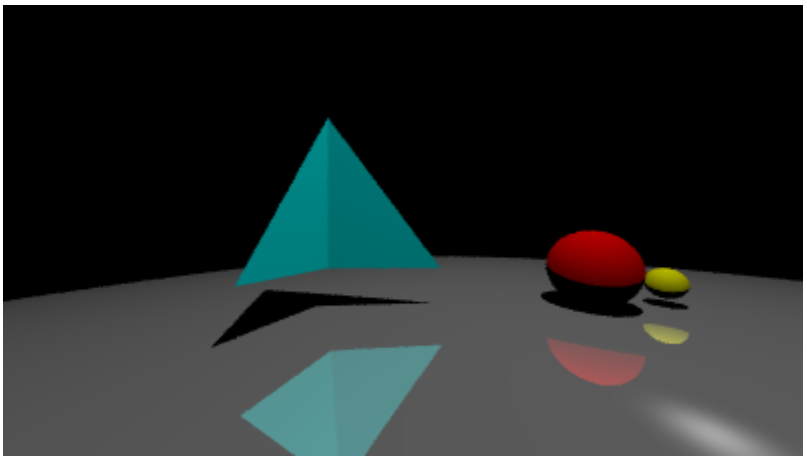
The class *vec3* (*vec3.h*) is a data structure used throughout the raytracing project. Most of the content in this class was from the source "Ray Tracing in One Weekend."

7 Bugs

We attempted to implement a tetrahedron; however, only 2 of the three sides rendered properly, so we opted to leave it out of the demo image. The tetrahedron code is in the *tetrahedron* class (*tetrahedral.h*) and like the *sphere* class, it is also a child of the *hittable* class.

	Vertices	Color (R, G, B)	Glaze
Tetrahedron	(-3.5, 2, -1) (-4, -0.0, 0) (-3.5, -0.0, -1) (2, -0.0, -1)	(0, 1, 1) [turquoise]	False

Below is the output of the demo image with the tetrahedron.



Project Sources

[1] “Ray Tracing in One Weekend.” raytracing.github.io/books/RayTracingInOneWeekend.html

[2] Peter Shirley and Steve Marschner. 2009. *Fundamentals of Computer Graphics (3rd. ed.)*. A. K. Peters, Ltd., USA.