Giovanni Perez Colon

Simple First-Hit Ray Tracer

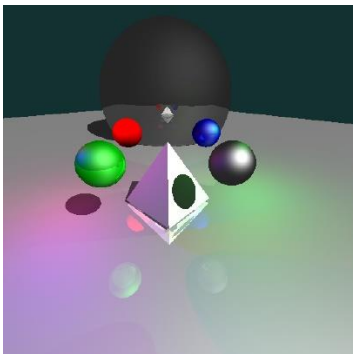
Report

1 Scene

The scene I created contains five spheres, one tetrahedron, and one plane comprised of two triangles. The spheres are implemented using my *Sphere* struct, the tetrahedron is created using four triangles from my *Triangle* struct, and the plane also utilizes the *Triangle* struct. These classes are all within my *types.h* header file. All the objects are instantiated within the render.cpp file above the for loop that renders each pixel. This code could've been more abstracted into a Scene class, but for time-constraint issues, *Spheres, Triangles,* and *Lights* are each separate types that can be instantiated and added into a list of their respective type. Each of these objects contain their own material properties, but will be discussed later in this report. The scene also contains four light sources: one directional light and three point lights. Each of these lights are of a different color.

The first sphere is located at coordinates (-15.0, 5.0, -15.0) and has a radius of 10.0. This is a large reflective sphere meant to demonstrate the ideal specular reflection implementation. The other four spheres are at Y position 4.0 and are each 4.0 units away from the world origin at (0.0, 0.0, 0.0), cornering this point.

The tetrahedron is located right above the world origin. A rotational matrix was implemented to allow for the object to rotate around the Y axis. The four triangles that comprise this object were created using five vertices, then specifying the order of these vertices in a clockwise direction to get an accurate surface normal. The direction is clockwise instead of counter-clockwise because the normal direction is flipped while calculating the surface shading.



2 Camera

This project contains both an orthographic and a perspective camera. I created a Camera struct that contained a function for getting a ray at a given pixel. There is a Boolean to toggle whether it uses the perspective ray calculation or orthographic calculation. The struct specifies the camera basis, position, perspective distance, orthographic scale, and max view distance.

2.1 Orthographic Camera

When the camera is specified as an orthographic camera, it shoots parallel rays in the -W direction, assuming a UVW basis that correlates with XYZ. These rays differ in their origin via the following formula:

- Ray's origin: $\mathbf{e} + u\,\mathbf{u} + v\,\mathbf{v}$
- Ray's direction: $-\mathbf{w}$

UV maps the image to a normalized coordinate system, allowing for easy resolution change.

2.2 Perspective Camera

When the camera is specified as an perspective camera, it shoots rays from the camera position, but all of their direction vectors are determined by the perspective distance, u, and v. I used the following formula to implement this camera:

- Ray's origin: $\mathbf{e}$
- Ray's direction: $-d\,\mathbf{w} + u\,\mathbf{u} + v\,\mathbf{v}$

3 Shading

The types header file contains the shading functionality under the Material class. Important variables in the Material class include *diffuseColor, ambientColor, specularColor, diffuseIntensity, ambientIntensity, specularIntensity, metallicFactor,* and *phoneExponent*. The colors and intensities are the values of their respective shading type. Phong exponent is used for the Blinn-Phong shading and metallic factor is the mirror coefficient for ideal specular reflections.

Lighting is implemented in the types file with the *Light* struct. Both directional lights and point lights share a position and direction, however, a bool *isPoint* is used to tell the shader function whether to treat the light as directional or point, or rather weather to acknowledge its position when calculating the vector pointing towards the light. Lights also contain a baseColor, which is a vector 3 of floats from 0.0 to 1.0 representing an RGB value. I should mention now that RGB colors are specified as vector 3 floats normalized from 0.0 to 1.0 when used in calculation. Doing this allows vector multiplication to act as color mixing. There are shading algorithms implemented in this code, ambient, diffuse, and specular.

The Material struct contains the function for calculating the color of a certain ray. The function takes a vector 3 surface normal, the camera ray, an integer specifying the number of light bounces for ideal specular lighting (glazing) (will be discussed in section 5), a reference to an intersection point, and a list of all lights, sphere, and triangles in the scene. The shading is done in five steps: ambient, diffusion, specular, shadows, and glazing. Each of these steps adds to a vector 3 called *finalColor* which is clamped to 1.0 at the end of the function to prevent colors higher than 1.0f.

3.1 Ambient Lighting

Ambient lighting is used to emulate the look of indirect lighting. Because of the complexity of global illumination and the implementation of indirect lighting, it was simpler to implement ambient lighting.

For ambient lighting, I simply multiplied the specified ambient color and the ambient intensity and then added it to *finalColor*. This is equivalent to the following function:

$$L_a = k_a I_a$$

Where *ka* is the ambient color and *Ia* is the ambient intensity. When instantiating a material, I found it best to specify the ambient color to be the same as the diffuse color.

3.2 Diffuse Lighting

Diffuse lighting is the first type of lighting to be affected by light sources. It is view independent and can be determined by the direction of each light source and the surface normal of the point that the camera ray intersected. I used the Lambertian shading function:

$$L_d = k_d I \max(0, \vec{n} \cdot \vec{v}_L)$$

Where *kd* is the diffuse color multiplied by the diffuse intensity, *I* is the light color multiplied by the strength of the light source, **n** is the normal vector of the intersection point, and **l** is the vector point towards the light source. Both **n** and **l** are normalized before calculation. Once diffuse is calculated, it's added to the *finalColor* vector.

3.3 Specular Lighting

Specular lighting is used to display the highlights of a shiny surface. I implemented the Blinn-Phong specular function, which relies on an exponent known as the *phongExponent* to determine how "tight" the highlight is. This lighting is view dependent which means that the camera direction vector has to be used during the color calculation. I used the following formula:

$$k_s I \max(0, \vec{n} \cdot \vec{v}_H)^n$$

Where *ks* is the specular color multiplied by the specular intensity, *I* is the light color multiplied by the light strength , **n** is the normal vector from the point of intersection, and **vh** is the vector representing the bisector of the vector point towards the camera and the vector pointing towards the light. This bisector is calculated because it is more efficient than traditional phong shading. Finally, the exponent is the *phongExponent*.

4 Shadows

Initially, I wasn't entirely sure how to go about shadows. I then realized that a shadow is simply where the light is non-existent. If you treated the lights like a camera, it could never see the shadow it casts, however, the light isn't a camera. I found that if you shoot a ray from the current intersection point in the direction towards the light, then iterate through all object lists to check if this new ray intersects those, you can determine whether the light from the light source is being blocked, or rather, if a shadow is being

casted. Shadows give the scene far more depth, but the drawback lies in performance. Because shadows are iterating through all of the objects again for every point that the camera ray hits, it takes longer to render an image. I didn't reference any particular function, instead I just shot another ray.

5 Glazed

Ideal specular reflection, or glazing, was the final step in shading. To implement this, I added a loop that ran for the number of light bounces specified. I then recursively called the shader function, subtracting light bounces by one so it doesn't run infinitely. Glazing gives a material a metallic reflection. The metallic factor is used to specify how intense the reflection of a given material is.

6 Animation

I created a short 150 frame, 512 x 512, five second animation that moved the camera along the z axis while rotating it to focus on the tetrahedron and while changing its perspective distance. Changing the distance allowed it to zoom in and out.

My sources were the textbook, the classroom, and https://www.scratchapixel.com/index.html.