Assignment 2 : The Path of Light

This coursework is based on Ray Tracing in One Weekend Book Series, Scratchapixel, some youTube channels and some github projects. Below is my workflow.

1. Complete parser
   First, complete the definition of the necessary structures, such as RayHitStruct, which includes the structures for Ray and Hit.

   In the Ray structure, define two objects, origin and direction, representing the origin and direction of the ray, respectively. This will facilitate the definition of the ray equation p(t) = origin + t * direction for later use.

   In the Hit structure, record the coordinates of the intersection point, the value of t in the ray, the normal vector, the number of bounces, and the shape.

2. Create camera and pointlight
   At the beginning, I started with the provided example code for the "camera" and found additional three variables in the JSON file, apart from the screen width, height, and field of view, which allow customizing the position of the camera. I then used the camera's position point, the target position point it is facing, and the up vector to calculate the tilt angle of the camera, and stored it in a 4x4 matrix.

   In Pinhole, the camera needs to be able to find rays on the coordinate system through ray casting, so it needs to be stored in a 4x4 matrix. The 4x4 matrix is designed based on the concept of homogeneous matrix, where the top left 3x3 matrix records the rotation angles of each axis, and the rightmost column records the displacement of the coordinates in space.

   The target position point in the lookAt function is first normalized, which also represents the z-axis. Then, using the cross product between the z-axis and the up vector of the camera, the x-axis is calculated. This step also requires normalizing all the position points. Finally, the y-axis of the camera is calculated using the cross product between the x-axis and the z-axis, and it is also normalized. Once all the calculations are done, the results are stored in the rotation matrix of the homogeneous matrix.

Next, write the point light source. In the point light source, the file records the position, intensity, and color of the light. Since the illumination of objects by the light rays requires the intersection of rays from the camera and the objects, the results cannot be displayed until the intersection points are generated. Due to hardware limitations, the results may not be displayed until the next step. Therefore, the objects will be displayed in the next step.

3. Ray-sphere intersection
   This step took a lot of time to research. I first wrote the code for the shape, so that all the shapes can inherit the functions inside the shape and read information from JSON files in this file. In each shape, there is color information, and I stored the color information needed for each object and passed it to the material. Since each object has a different color, it is more convenient to implement it in the material instead of implementing it in each object, which makes it easier to check for issues and reduce errors.

   For the Sphere, I implemented the formula for ray-sphere intersection. First, I calculated the solution (t) for the intersection between the ray and the sphere, which is a solution to a quadratic equation. I used the origin, direction, center, and radius of the sphere in the calculation of coefficients a, b, c in the ray equation.
   The formula is as follows:

$$t^2\vec{\mathbf{b}}\cdot\vec{\mathbf{b}}+2t\vec{\mathbf{b}}\cdot(\mathbf{a}\overrightarrow{-}\mathbf{c})+(\mathbf{a}\overrightarrow{-}\mathbf{c})\cdot(\mathbf{a}\overrightarrow{-}\mathbf{c})-R^2=0$$

   Next, the function uses the value of the discriminant of the quadratic equation to determine if the ray intersects with the sphere. If the discriminant is less than zero, it means the ray does not intersect with the sphere. Otherwise, if the discriminant is greater than or equal to zero, it means the ray intersects with the sphere, and the function returns the nearest intersection point, position, and normal vector.

   In the scene, the information for the background color is first extracted from JSON, and the information for the shapes and light sources is parsed. Then, in ray casting, the interaction between rays and objects is simulated, and the focal point of the rays with the objects is calculated.

   The generated image is shown below:

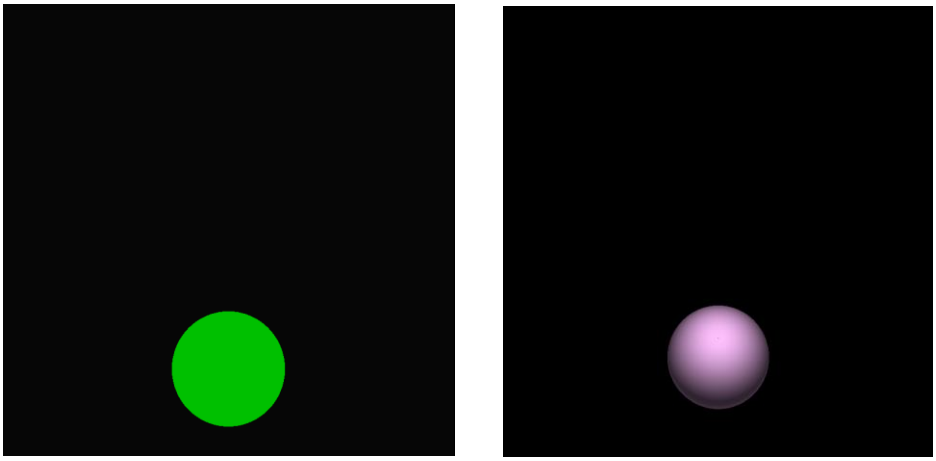

Figure 1 : ray-sphere intersection

Initially, there were some mistakes in the code, resulting in slight color errors in the output(Figure 1 left). However, after subsequent corrections, the colors are displayed correctly, as Blinn-Phong shading has been added(Figure 1 right).

4. Ray-plane intersection

   In this intersection, four points are used to calculate the normal vector of the plane. Firstly, the normal vector of the entire plane, Lo, is obtained by taking the cross product of two different edges. Then, using the following formula, the intersection point is calculated. ◦

$$t = -\frac{(l_0 - p_0) \cdot n}{l \cdot n} = \frac{(p_0 - l_0) \cdot n}{l \cdot n}$$

   If the normal vector Lo is less than 0 or the ray is parallel to the plane, it means there is no intersection with the plane, and the function returns an infinite distance and a zero vector as the normal vector.

   After obtaining the value of t, it is also necessary to ensure that the calculated intersection point falls within the range of the plane. Otherwise, the function still returns an infinite distance and a zero vector as the normal vector.

   The generated image is shown below:

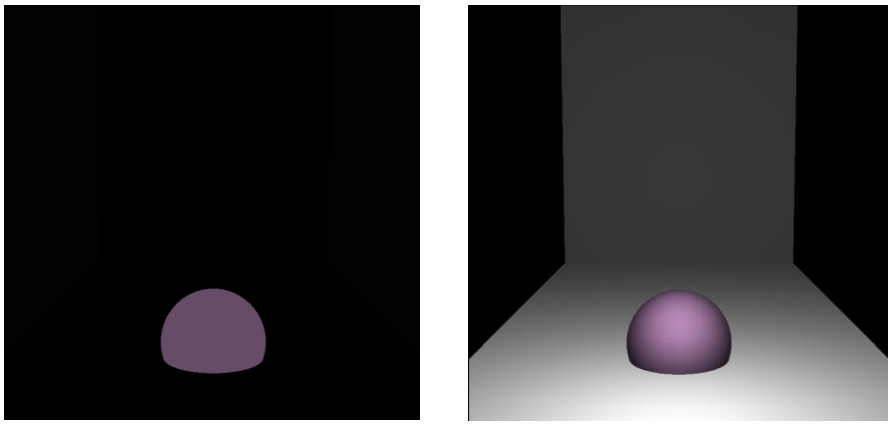

Figure 2 : ray-plane intersection

Initially, there were some mistakes in the code, resulting in slight color errors in the output(Figure 2 left). However, after subsequent corrections, the colors are displayed correctly, as Blinn-Phong shading has been added(Figure 2 right).

5. Ray-triangle intersection

   I used a similar method as with the plane to calculate the normal vector of a triangle.

   By taking the three vertices of the triangle, drawing two edges, and performing the cross product, I obtained the normal vector of the triangle. Then, I used the inside-outside pseudocode in here to determine if the coordinates are inside the plane. If they are, I returned the intersection point information, otherwise, I returned an infinite distance and a zero vector for the coordinates.

```
Vec3f edge0 = v1 - v0;
Vec3f edge1 = v2 - v1;
Vec3f edge2 = v0 - v2;
Vec3f C0 = P - v0;
Vec3f C1 = P - v1;
Vec3f C2 = P - v2;
if (dotProduct(N, crossProduct(edge0, C0)) > 0 &&
    dotProduct(N, crossProduct(edge1, C1)) > 0 &&
    dotProduct(N, crossProduct(edge2, C2)) > 0) return true; // P is inside the triangle
```

Figure 3 : inside-outside pseudocode

The generated image is shown below:

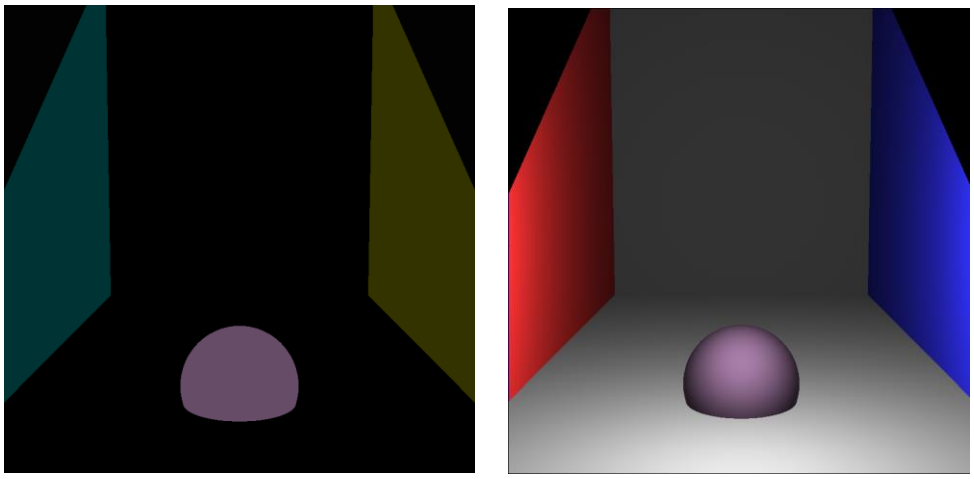

Figure 4 : ray-triangle intersection

Initially, there were some mistakes in the code, resulting in slight color errors in the output(Figure 4 left). However, after subsequent corrections, the colors are displayed correctly, as Blinn-Phong shading has been added(Figure 4 right).

6. Ray-mesh intersection

   The important part here is how to read a PLY file. First, we need to parse the header information in the file to obtain the number of vertices and faces. Next, the function loops through the file to read the vertex information. Each vertex is represented by three floating-point numbers for the x, y, and z coordinates. The function multiplies the read vertex coordinates by a scaling factor and adds an offset to create a new Vec3f object, which is then added to the topPoint container.

   Next, the function continues to loop through the file to read the face information. Each face is represented by an integer that indicates the number of vertices in the face. The function then reads three integers that represent the indices of the three vertices of the face. Based on these indices, the corresponding vertices are found in the topPoint container, and a new Triangle object is created and added to the vectorFace container.

   A TriMesh is a pattern composed of many small triangles, so after reading the file, we can use the intersection function of Triangle to find the intersection points of the object.

   The generated image is shown below:

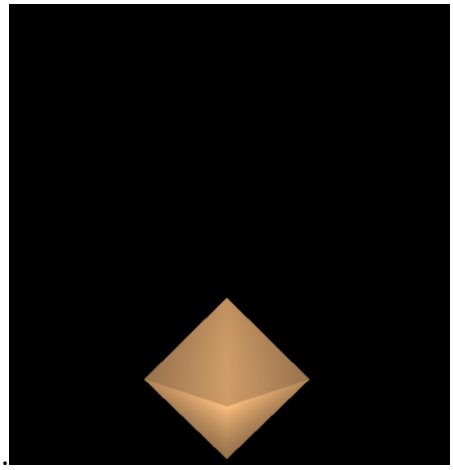
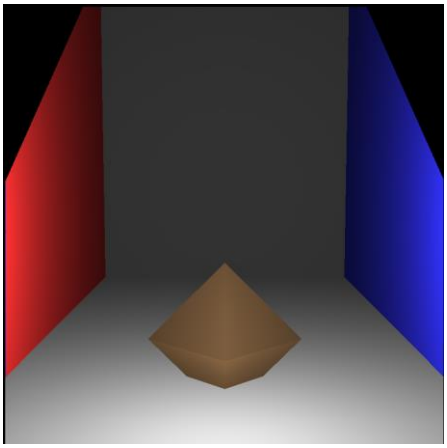

Figure 5 : ray-mesh intersection

This is the pattern I rendered, and I added it to the scene.

7. Blinn-Phong Shading + Reflection

   First, calculate the direction vector of the incoming light ray, which can be obtained by subtracting the position of the light source from the intersection point.

$$\vec{l} = \frac{\vec{p} - \vec{L}}{\left\|\vec{p} - \vec{L}\right\|}$$

   Where p_hat is the position of the intersection and L_hat is the position of the light source

   Next, calculate the direction vector of the reflected light ray, which can be obtained from the normal vector and the direction vector of the incoming light ray

$$\vec{r} = \frac{2\vec{n}(\vec{n} \cdot \vec{l}) - \vec{l}}{\left\|2\vec{n}(\vec{n} \cdot \vec{l}) - \vec{l}\right\|}$$

   Where n is normal vector

   Then, compute the direction vector of the viewing ray, i.e. the vector from the observer's position to the intersection point, which can be obtained by subtracting the position of the camera from the intersection point.

$$\vec{v} = \frac{\vec{c} - \vec{p}}{\|\vec{c} - \vec{p}\|}$$

Where c is position of the camera.

Next, calculate the direction vector of the halfway vector, which is the average of the direction vector of the incoming light ray and the direction vector of the viewing ray

$$\vec{h} = \frac{\vec{l} + \vec{v}}{\left\|\vec{l} + \vec{v}\right\|}$$

The formula for calculating the diffuse reflection intensity

$$I_d = k_d(\vec{l} \cdot \vec{n})$$

where Kd is the diffuse reflection coefficient and n is the surface normal vector. The diffuse reflection intensity is higher when the angle between the incoming light ray and the surface normal is smaller.

The formula for calculating the specular reflection intensity

$$I_s = k_s(\vec{r} \cdot \vec{v})^{n_s}$$

where ks is the specular reflection coefficient and ns is the shininess coefficient. The specular reflection intensity is higher when the angle between the reflected light ray and the viewing ray is smaller.

Reflection

When designing the reflection of light, the ray type defined by Ray is used to determine whether it is the primary ray type, denoted as PRIMARY, indicating that it is the first ray being traced (rather than a reflected ray), and if the object has reflective properties (kr > 0), then the direction and corresponding color of the reflected light are calculated.

The reflected ray is defined as SECONDARY, and another ray is cast recursively to obtain the reflected color. The reflected color is then added to the intensity

vector. Finally, all the light sources are summed up to get the overall lighting of the scene.
The formula for the final color is as follows:

$$Color = diffuse + specular + reflection$$

The generated image is shown below:

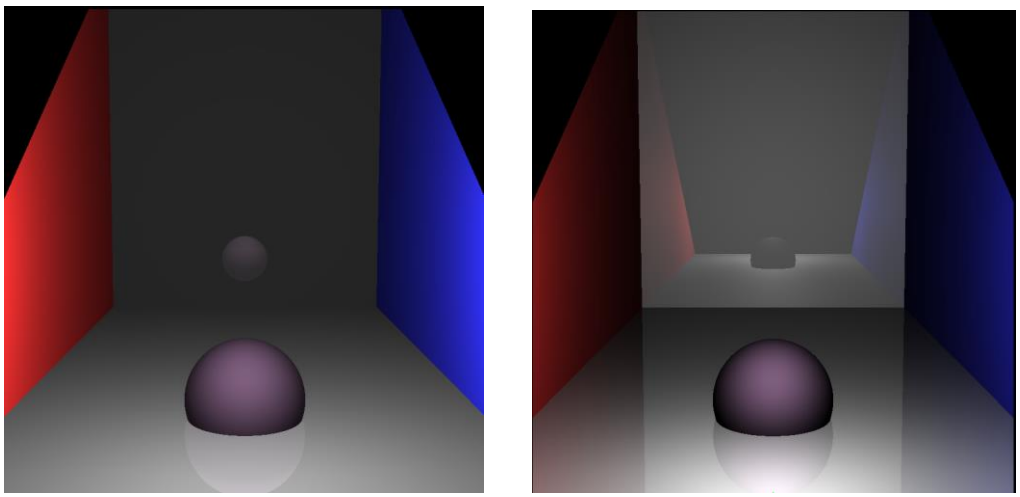

Figure 6 : Blinn-Phong Shading + Reflection

Due to a coding error in the beginning, only spheres were able to reflect onto the upright mirror. However, after making modifications, objects can now be successfully reflected onto the upright mirror. However, there seems to be an excess of background color, and a solution has not been found yet.

8. Texture mapping
he basic principle of texture mapping is to map a 2D texture onto the surface of a 3D object. During the rendering process, each surface of a triangle or polygon is treated as a plane, and the texture is mapped onto the surface based on the coordinates of the vertices and the texture coordinates (also known as UV coordinates). Typically, texture coordinates are a 2D coordinate system where U and V represent the position of the texture in the horizontal and vertical directions, respectively, with a range usually between 0 and 1.

The formula for U and V coordinates of the material on a sphere are as follows:

$$tu = asin(Nx)/PI + 0.5 \quad tv = asin(Ny)/PI + 0.5$$

For planar surfaces, the material is mapped based on its width and height. For triangles, the longest edge is used as the base, and the height is calculated using the Pythagorean theorem to obtain the mapping result.

Afterward, the material is applied using the Blinn-Phong shading formula.

The image below shows the final result.

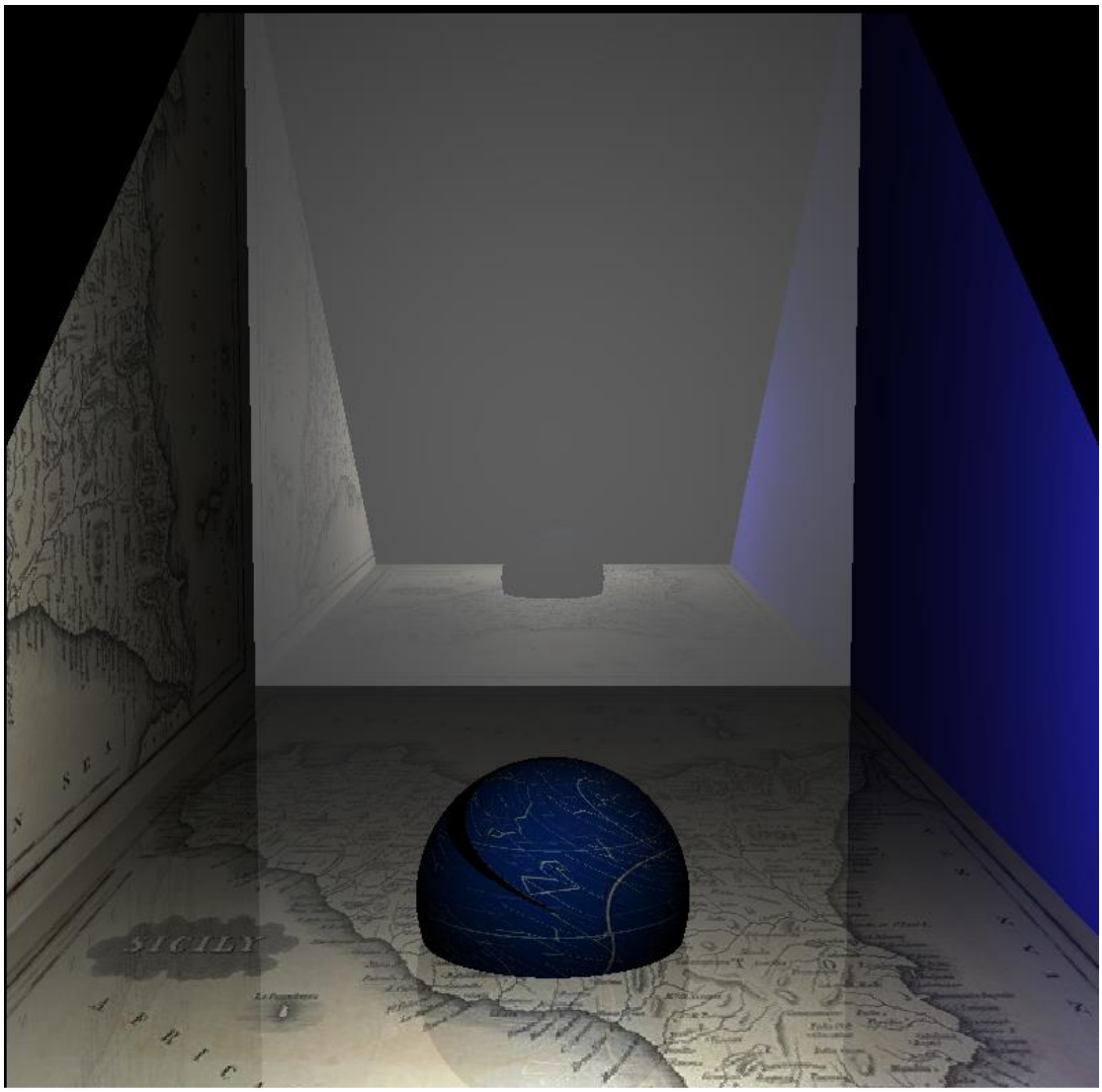

Figure 8: Texture mapping

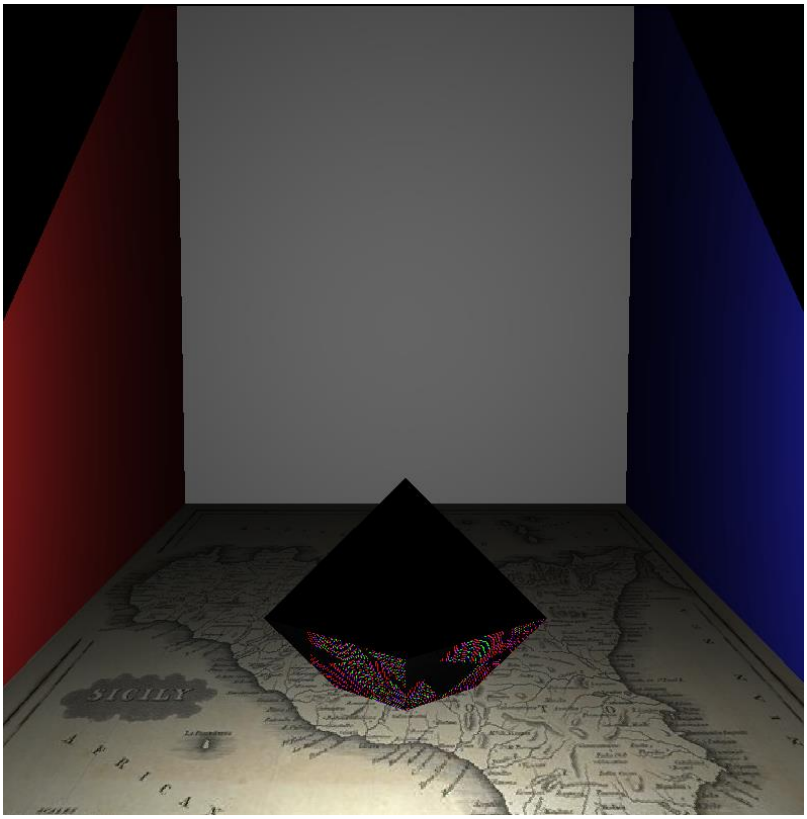

Figure 9: Texture mapping with Mesh

Although I have also tried using texture mapping on trimeshes (Figure 9), I followed the aforementioned approach for triangles and obtained only this result. I estimate that I need to first calculate the total surface area and then divide it by the area of each triangle to obtain the correct result.

Conclusion

Due to my limited skills, I have only been able to complete the aforementioned steps. Although I have researched BVH extensively, I still haven't been able to implement it successfully.

Overall, it has been an interesting learning experience.