Iris Corbalan, Mar Moreno, Àlex Pachón

LAB 1: Whitted Ray Tracing

Task 1: Painting the Image

For the first task, we will modify the *existing PaintImage()* method in the main function, which is responsible for painting the image. Currently, the image is painted using random colors. We will replace the *random_color* vector with a new color, which will be calculated using the formula provided in the lab instructions:

$$color = RGBColor(\frac{x}{width}, \frac{y}{height}, 0)$$

In this formula:

- *col* and *lin* represent the *x* and *y* coordinates of the pixels.
- *resX* and *resY* represent the *width* and *height* of the image.

Once this modification is made, the output image looks like Figure 1.

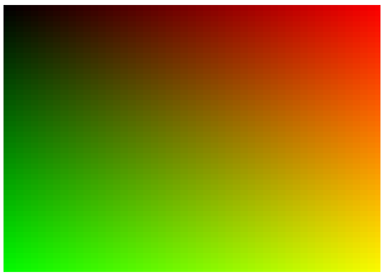

Figure 1. Image painted

Task 2: Implement the Intersection Integrator

In this task, we will modify the *PaintImage(film)* function from the previous exercise and integrate it with the *raytrace()* function in the main code. Instead of generating a random image, we will use ray tracing to compute the color for each pixel. This is done by applying the *shader* (as indicated in the comments).

By setting up a loop for ray tracing, where we compute the color of each pixel using the shader, the resulting image will match the one shown in Figure 4 of the exercise.

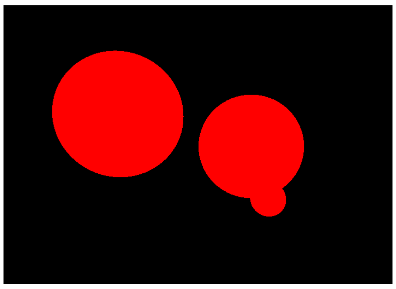

Figure 2. Output image on Task 2

Task 3: Implement the Depth Integrator

To implement the depth integrator, we need to modify the *depthshader* to consider intersections between the rays and objects in the scene. This will help us to define which objects are covered by other objects and which are closest to the camera.

When a ray intersects an object, we check all intersections and find the closest one to the camera. Once the closest intersection is identified, we compute the distance from the intersection point to the camera. This distance will be used to shade the object.

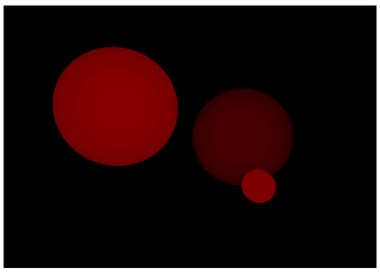

Figure 3. Output image on Task 3

The shading will be based on the formula provided in the exercise, where the computed color corresponds to the distance. To normalize the distances, we use a maximum distance of 7, as recommended by the teacher. This ensures that objects closer to the camera are shaded darker, while objects farther away approach a lighter color. The result is shown in Figure 3.

Task 4: Implement the Normal Integrator

In this task, we need to switch the shader being used to the normalshader, similar to the previous exercise with the depth shader. This requires creating the necessary files for the normalshader.

Since all integrators inherit from the shader class, we include its header and follow the same structure, making appropriate name changes for clarity and convenience. The goal here is to visualize the normals of the scene while still considering the depth of the objects, so we will structure the normalshader similarly to the depthshader we implemented earlier.

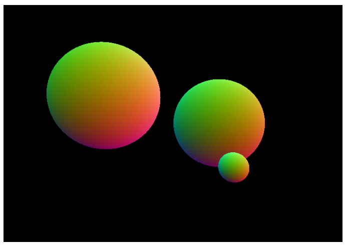
Figure 4. Normal Integrator Image

When a ray intersects an object, we find the closest intersection point. Instead of computing the distance as we did with the depth shader, we will compute the normal vector at the intersection point. This is done by using the normal attribute from the intersection class. Once we extract the normal vector, we follow the formula provided in the exercise to compute the final color based on the normals.

Task 5: Implement Whitted Integrator

In this part of the exercise, we will implement the Whitted Integrator to simulate more complex light interactions, such as reflections and refractions. To begin, we need to modify our scene, the Cornell Box, by adding materials that can showcase these effects (reflective and transparent surfaces). Similar to previous exercises where we created a new integrator, we will now develop the Whitted integrator shader and update the ray tracing function to use this new shader. This will enable us to trace reflection and refraction rays, as well as calculate shadows, leading to more realistic renderings.

5.1 Phong Material

To compute the phong function, we need to complete the whitted integrator and complete the *computeColor()* formula. Since we are considering the depth of the objects, the closest intersections will be used to calculate the colors of the scene correctly.

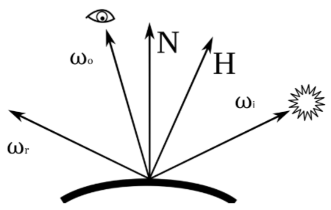

Figure 5. Vectors representation

1. Loop through Multiple Objects: In this scene, we have multiple objects stored in a list. We will loop through each object to compute the color for all of them based on the closest intersection and on Figure 5.

2. Vector Calculations: For each intersection, we need to calculate the following vectors:
- wi: The vector from the light source to the intersection point.
- n: The normal vector at the intersection point.
- wo: The outgoing vector, which represents the ray direction from our camera.

3. Phong Reflection Model: With the vectors (wi, n, wo) known, we move on to the Phong reflection model. In Exercise 4.5.1, we have already implemented the necessary Phong parameters:
- Ks: The specular reflection coefficient.
- rho_d: The diffuse reflection coefficient.
- alpha: The shininess exponent.

4. Reflectance Calculation: The *getReflectance()* function in the Phong shader is responsible for computing the final color. We compute separately:
- Diffuse part: Calculated based on the normal vector n and the incoming light direction wi.
- Specular part: Includes the reflection vector wr, which is computed from the normal vector n and the incoming direction wi. The specular highlight is computed using Ks and alpha.

The final color is obtained by adding the diffuse and specular components.

5.2 Direct Illumination with Point Lights

Once we have our phong function completed and the vectors defined in our whitted integrator, we call the phong function. To end up with our desired result shown in figure 7, we need to create a new ray defining its characteristics as its origin being the intersection point, the direction being the same as wi and a maximum being the exact distance between the origin of the ray to the position of our light, this is to understand if there is an intersection and a shadow is needed to be computed. If there is an intersection, then there is no visibility term. Then by following the first formula, we get the figure number 6. But it's still dark since we did not add the ambient term. We create an ambient constant by creating a new vector (we have chosen values of 0.15 so that the image is brighter but not too much) and our diffuse coefficient is created by getting the diffuse reflectance of the material. Then the result can be compared as follows:

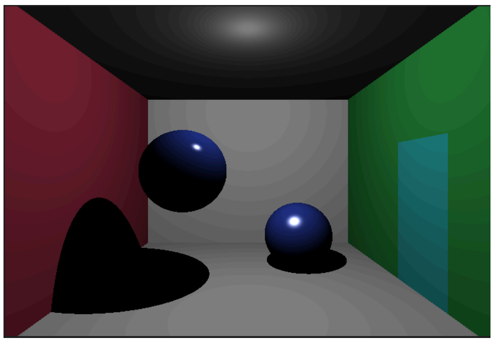
Figure 6: Image with Phong

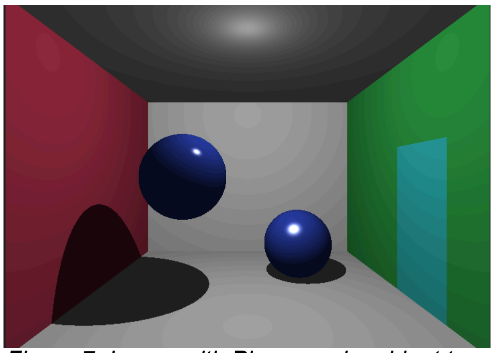
Figure 7: Image with Phong and ambient term

5.3 Mirror Material

To render the image with a mirror, first, we create a *Mirror* class in the Material folder, which is similar to the *Emissive* class. The key difference is that the *hasSpecular()* function will return *true*, while all other boolean functions return *false*. Next, we update the main program to assign the *Mirror* material to the square object.

Once the material is set, we compute its color using the *computeColor()* function in the *WhittedIntShader* class. Here, we check if the shape is a mirror by calling *hasSpecular()*. If true, we calculate the reflection direction wr using the formula:

$$\omega r \;=\; 2(\omega i \cdot N)N \;-\; \omega i \quad \text{(Eq 2)}.$$

Then, we create a reflection ray with the computed direction and call computeColor again with the reflection ray to get the reflected part of the mirror image.

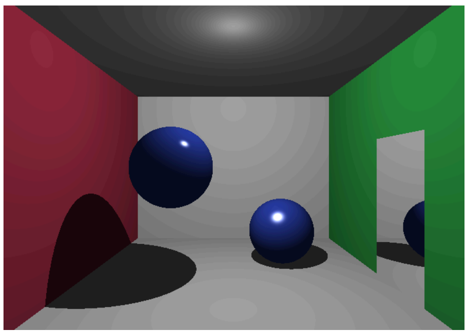
Figure 8: Image with mirror

5.4 Transmissive Material

Similarly to how we created the mirror material, we create a *Transmissive* class, where only the *hasTransmissive()* function returns *true*. If more boolean functions were set to *true*, it could lead to a stack overflow due to excessive recursive operations. To make the transmissive material visible, we assign it to the second ball and complete the Whitted integrator when the material returns *true* for *hasTransmissive()*.

To avoid total internal reflection, we change the value of the ratio to its inverse and change the sign of the normal. Then if the value of inside the square root is positive (otherwise it would not be able to be computed) we calculate the transmissive refraction. If it's not, we compute the reflection direction. Once all these steps are completed, the result in figure 9 is produced.

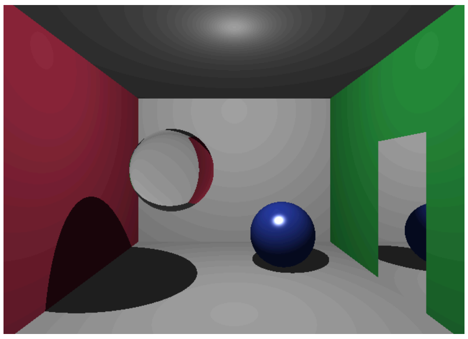
Figure 9: Completed image