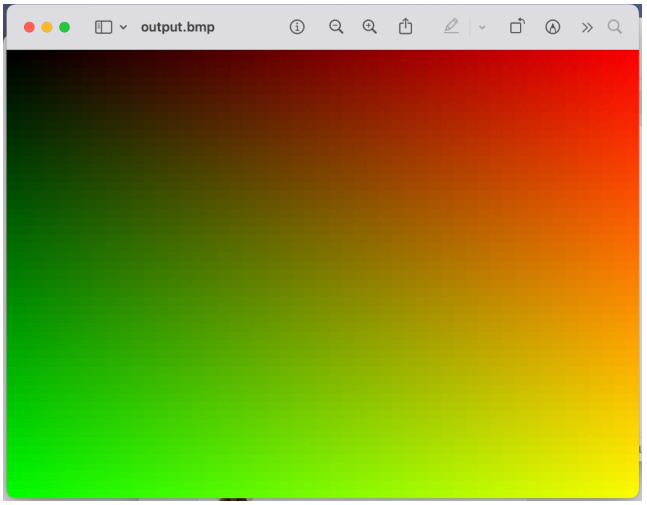
Viktoria Olmedo - 252293
Javier Portabella - 251515

ADG

REPORT LAB

EXERCISE 1

In the first exercise we obtained as a result the following image:

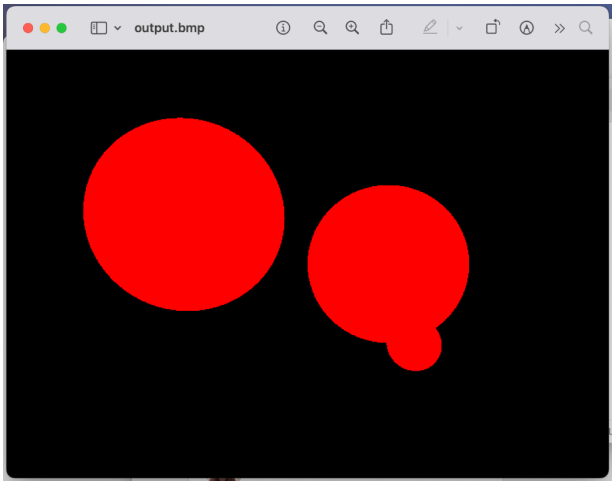


At the beginning of the lab, the output image was painted with random colors for each pixel (random noise), but we modified the rendering loop to assign colors based on pixel coordinates. We calculated the color values by creating a relationship between the x and y axis and the colors. This created a gradient effect where the red component varied along the axis x and the green component varied along the axis y, while the blue component remained zero.

EXERCISE 2

In the second exercise we started using the sphere objects in the image by implementing the IntersectionShader that just returns a specified color if an intersection is found and another color (background color) otherwise.

The result obtained in this exercise is the following:

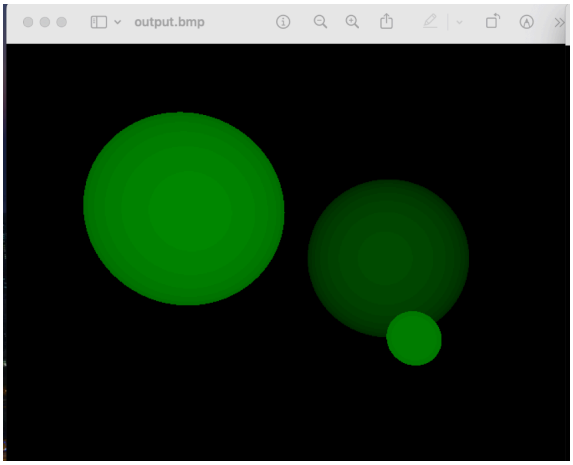


Viktoria Olmedo - 252293
Javier Portabella - 251515

ADG

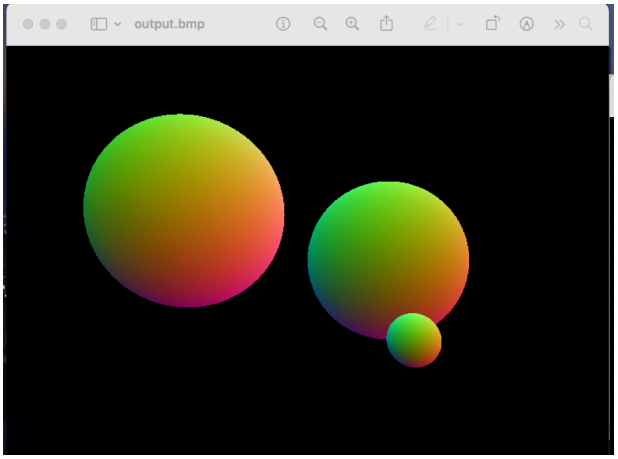
EXERCISE 3

In this step, we developed a DepthShader to visualize the distance between the camera and the objects in the scene. The shader calculated the color intensity based on the distance to the intersection point.



EXERCISE 4

In this exercise we applied the given formula to get the color of each pixel using the normals of each point of the sphere. We had a problem with the colors at the beginning because we were applying the formula in the wrong way (with a parenthesis in the wrong place). At the end, we solved the problem and got the following result:



Viktoria Olmedo - 252293
Javier Portabella - 251515

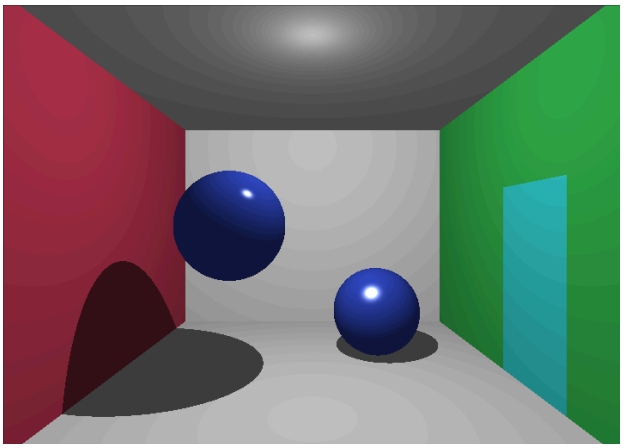
ADG

EXERCISE 5.1

In this exercise we had to complete the Phong class, which allows the light interactions with surfaces. We just applied the formula to the class.

EXERCISE 5.2

Now we had to create the WhittedIntegrator class, which had, like the other shaders, the ComputeColor() function. We first check if the ray intersects with any objects in the scene, if there is no intersection, we return the background color. If there is an intersection, we retrieve the surface normal at the intersection point and compute the direction of the ray wo. We iterate through all the light sources in the scene and for each one we compute the direction wi from the intersection point to the light source and construct a shadow ray to check for occlusions. If the shadow ray doesn't intersect any objects (meaning the point is lit), it calculates the light's contribution to the color based on the material's reflectance properties and the angle between the normal and light direction.
For the ambient term, we put the ambient constant to 0.3, which we felt was the perfect amount of brightness.
In this function we had a problem with the distance used to create the shadow ray. We were first taking the length of wi, which is the normalized distance between the intersection point and the light source, so because it was normalized, the length would always be one, so that was causing trouble. We created `double distanceLight = (lightSourcePos - its.itsPoint).length();` to pass as a parameter to the shadow ray.



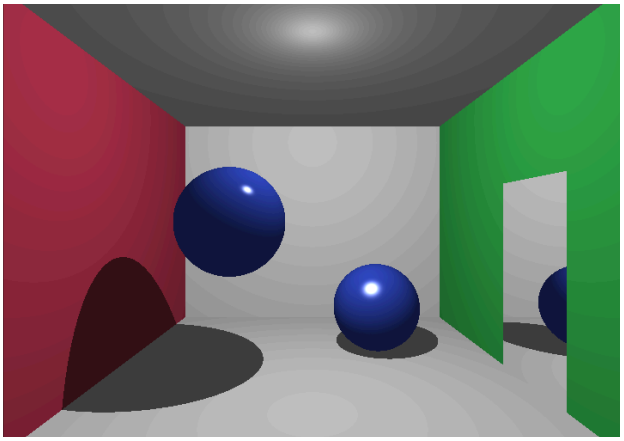
EXERCISE 5.3

For this exercise, we first saw that two new *ifs* were needed applying the functions given: *hasSpecular()* and *hasDiffuseOrGlossy()*. The first checks whether the material reflects light in a specular way. If the material has indeed specular reflection, we will apply the formula to compute w_r and then the reflected ray with the result. To generate this one, we need to enter as a parameter the maximum distance of the ray but, as this needs to be infinite (so that it can be launched "infinitely far"), we have decided to fix a very big number as 1·10^6.

The second *if*, with the second function given, contains the same code as the last exercise and generates the rays for the glossy or diffuse reflections.

To continue, we have modified the *main*. A new instance of a mirror material needs to be created. We use it to change the material of the rectangle on the right of the image from *cyandiffuse* to *mirror*. In order to make this instance we first need to create a new class that inherits from Material and just fixes the bool functions to a given value depending on the properties of a mirror.

The unique difference visible in the result with respect to the last exercise, is the mirror that has appeared in the right side of the image:



EXERCISE 5.4

This last exercise we have tried to modify the ComputeColor function in whittedintegrator.cpp again. After writing the last if with the hasTransmission() function, we look at the conditions that the document advises us to look at (whether the ray is entering to the sphere or not).

We have created a new class called transmissive.h as we did with the mirror.h. We have done it to generate the instance of the transmissive material in the main. Inside this new file there is the getIndexOfRefraction() and the getReflectance() but this one only returns a zero vector because there is no reflectance here.

The problem we have had with this exercise is when changing the blue material of the spheres into the transmissive materials. We don't understand why we get a lot of exceptions when running the code.