Assignment 2 report

Giulia Benvenuto

April 6, 2022

Environment:

- Windows 11
- Visual studio 2022 17.1

1 Sphere and Parallelogram Intersection Functions

I use my implementation of them from assignment 1 readjusting them according to the structure of this assignment.

2 Ex.1: Field of View and Perspective Camera

I changed "*scale_y*" and "*scale_x*" computing at first "*scale_y*" using the trigonometric formula for the right-angled rectangle and then "*scale_x*" using the variable "*aspect_ratio*".

```
double aspect_ratio = double(w) / double(h);
double scale_y = scene.camera.focal_length * tan(scene.camera.field_of_view / 2.0);
double scale_x = scale_y * aspect_ratio;
```

Figure 1: Calculation of scale_y and scale_x

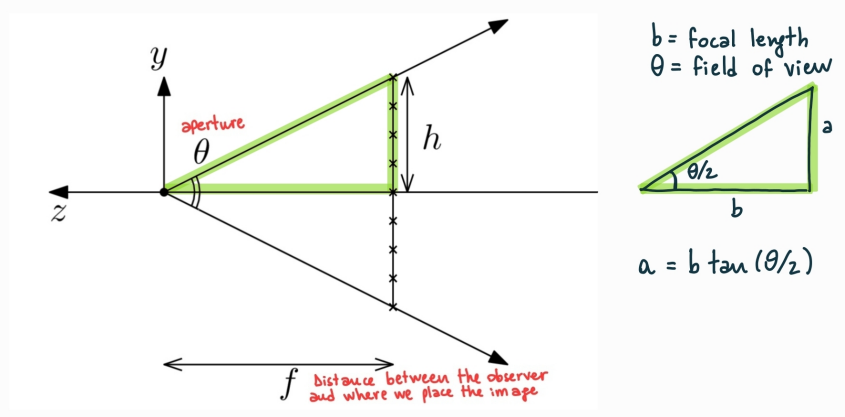

Figure 2: Reasoning

3 Ex.2: Shadow Rays

I cast a shadow ray to determine if the light should affect the intersection point. The origin of the ray is the position of the intersection point and the direction is the one towards the light. I also set an "*offset*" equal to 0.0001, along the light direction from which the ray has to start because otherwise the ray intersect the object from which it is cast. Without an offset I got that many pixels were black and the image had a strange effect like the following image.

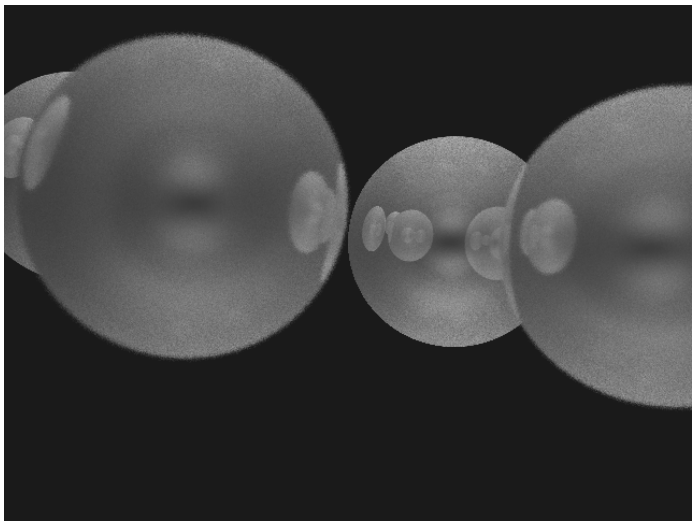

Figure 3: Shadow rays without offset

With an offset I got a well defined and brighter image like the following one.

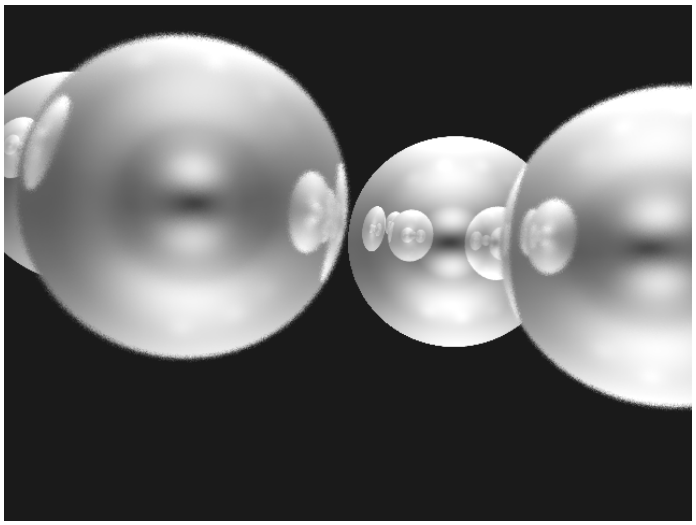

Figure 4: Shadow rays with offset

4 Ex.3: Reflection & Refraction

I computed the specular contribution. Ati first I calculated h and then I used it to calculate the specular coefficient to be placed inside the final shading equation.

I computed the color for the reflected ray by checking that the color inside the matrix called "*refraction_color*" was different from black (0, 0, 0) and that the number of bounce made by the ray was still bigger than zero. If the conditions were verified, r is calculated using this formula r = 2n(n·v)-v, then a ray with an offset is cast and the refraction color is computed.

To compute the refracted ray I checked the same conditions for the reflection case and I distinguished between the two cases, the first is when the ray passes from the air to the object in the scene, the second one is the opposite. Then I used the **refraction** function to compute the direction of the refracted ray using the Snell-Descartes law.

5 Ex.4: Depth of Field

I decided to cast 20 rays for pixel. I implemented the depth of field using a for loop over the number of rays. Inside it, if the camera is perspective I set the origin of the ray equal to the camera position and the position of the pixel equal to the position of the camera plus the given shift. Then I computed two random variables called "*rand_x*" and "*rand_y*", the first one is multiplied by the lens radius of the camera because it represents the distance from the origin of the ray and the second one is multiplied by 2 * pi because it represents the angle. Then I added these two quantities to the origin and I computed the direction for the ray. At each cycle of the for loop a ray is cast and at the end of this for loop I divided the variable C by the number of rays. I changed the focal length to see the result.

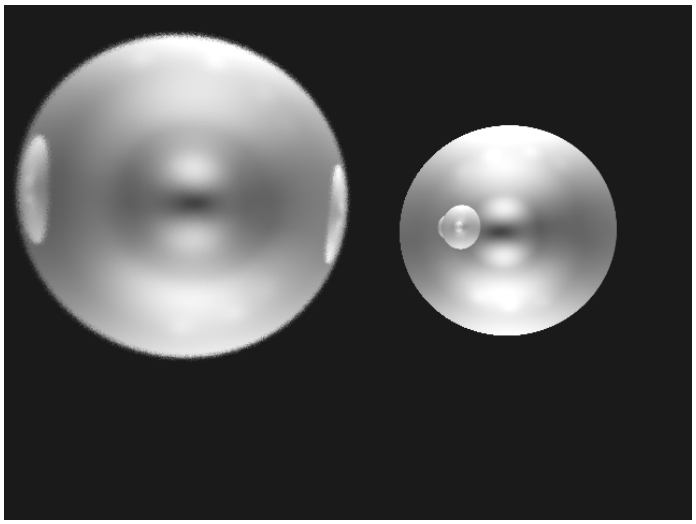

Figure 5: Focal length = 6.0, the sphere in the foreground is blurred, the one further away is sharp

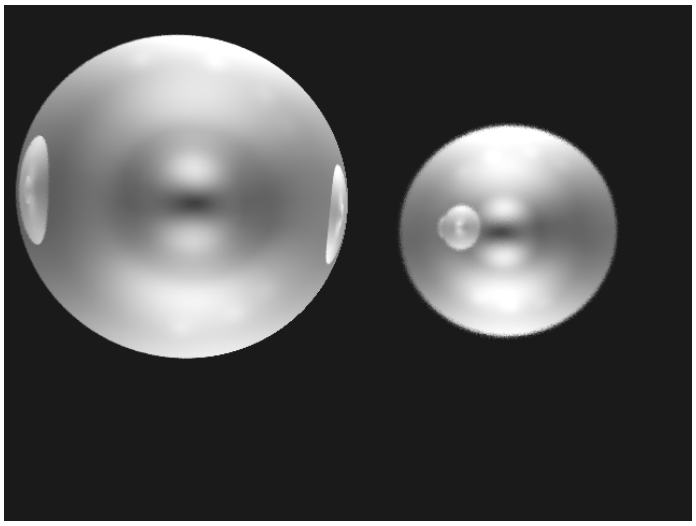

Figure 6: Focal length = 4.0, the sphere in the foreground is sharp, the one further away is out of focus.

6 Ex.5: Animation

I implemented the animation within a dedicated function called **gif_animation**. I decided to set the delay at 20 ms. I tried to move all the spheres far from the camera. The result is called **out.gif** it is inside the archive.

7 Note

In the data folder I used two scenes, the one called "*scene_1*" is colored and has a different number of spheres, only four. The one called "*scene_2*" is the original one, given with the code.