Rendering Project

This project is an implementation of simple physically-based rendering techniques. Upon execution, it creates several 512x512 images, showcasing the existing features, and stores them in the folder `/_results`. The following documents provides an overview of the methods employed as well as the obtained results.

Scene A scene is a set of geometric objects to be translated into an image. In order to create a background, we are going to use six spheres with large enough radii which bound an approximately box shape. A very first example to showcase a scene's general look can be seen in Figure 1.

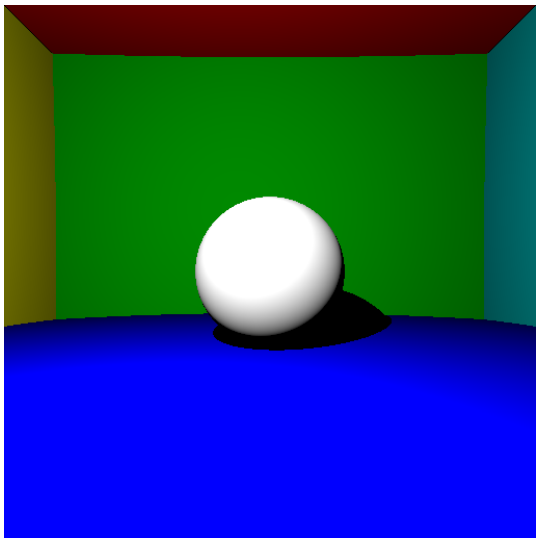
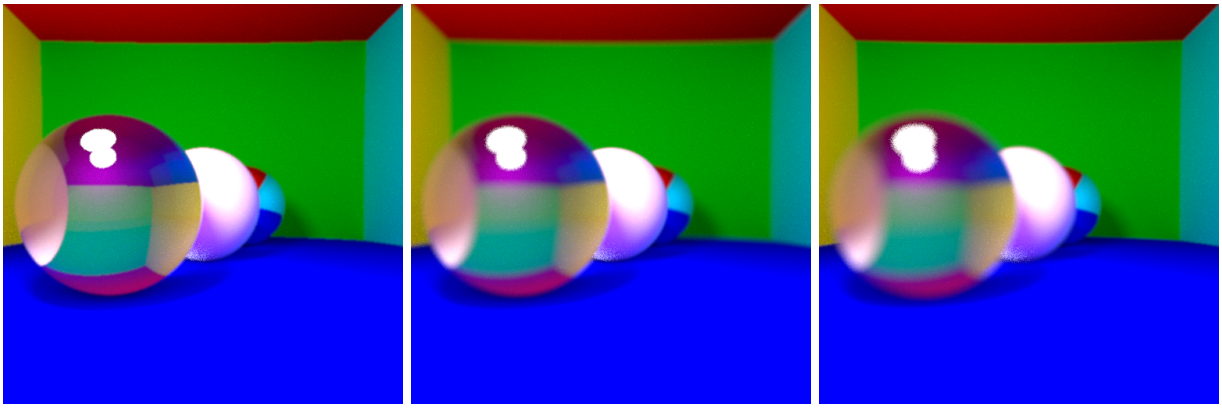


Figure 1: Basic scene

Camera A camera consists of a center Q and a virtual plane on which the scene will be projected. That plane can be characterized by an angle α. In all the images a camera with center $(0, 0, 55)$ and angle 60^o has been used.



(a) pinhole camera, no depth of field

(b) aperture with focus distance 50

(c) aperture with focus distance 100

Figure 2: Camera types

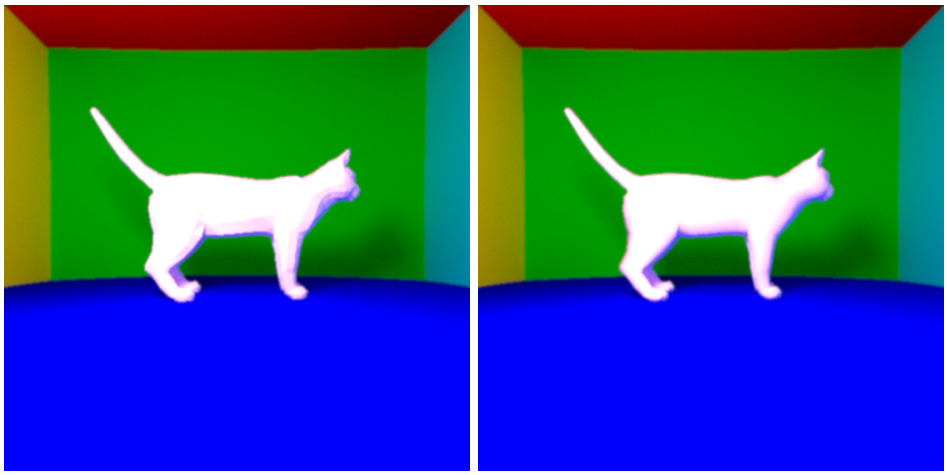
Two camera types are possible - the pinhole model, where the images are sharp at all distances and a circular aperture one, where the images are sharp only around a certain focus distance. To obtain the latter, we generate

arbitrary view points for which the focus distance would be the same in a disk around the actual center and average out the results from several runs. The difference these camera models make can be observed in Figure 2.
In the examples below, we will only use the pinhole camera model.

Rays Once our camera is set, we send rays from it throughout the scene, so that when considering the virtual display plane, a ray is sent towards each pixel.
In order to achieve a smoother picture, we avoid simply targeting the middle of each pixel, but instead utilize a Gaussian filter. It samples points surrounding each pixel center according to the Gaussian distribution and then averages the results. Unless otherwise specified, we are going to launch 1000 rays for each pixel.

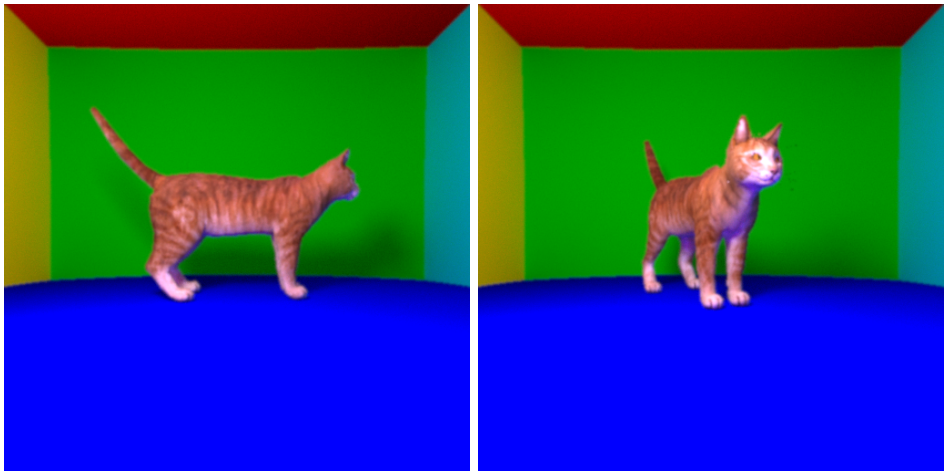
Geometries The rays travel through a scene consisting of objects of the class `Geometries`. Each has a color represented as a tuple $(red, green, blue)$, a surface type and the corresponding to it parameters.
All of our scenes are constructed with two types of geometric objects.

- **Sphere**
  The class `Sphere` inherits from `Geometries`. Every sphere is further characterized by a center and a radius.
- **Triangle mesh**



(a) no interpolation of normals (b) with interpolation of normals

Figure 3: Phong interpolation

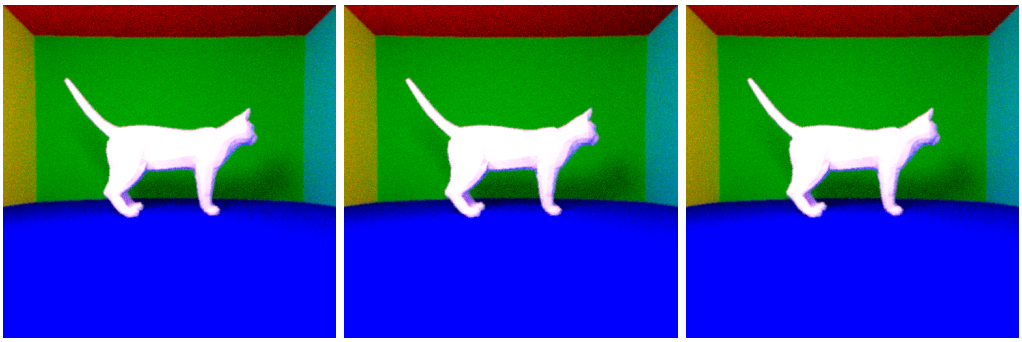


(a) model with texture (b) same model, rotated

Figure 4

The class `TriangleMesh` also inherits from `Geometries`. A triangle mesh object is read from a `.obj` file and stored as sets of triangle vertices, normal vectors and uv coordinates. The resulting model is then scaled, rotated and translated as necessary in order to fit with the rest of the scene. Before proceeding, the function `set_parameters` must be called. It allows the user to indicate if the set of normals should be used to obtain a smoother, more realistic image as shown in Figure 3 and if the set of uv coordinates should be used to sample texture from a provided file as shown in Figure 4.

Ray-object intersection In order to obtain an image, we have to note the intersections between rays and geometric objects. We proceed ray by ray, testing for an intersection between it and every geometric object contained in the scene. The algorithm varies depending on the type of geometry.
For spheres, we attempt to solve the quadratic equation $t^2 + 2t < u, O - C > + ||O - C||^2 - R^2 = 0$, where O is the origin of the ray, u is its direction and C and R are the sphere's center and radius respectively. If a positive solution exists, it corresponds to the distance between O and the sphere.
For triangle meshes, we can test for an intersection triangle by triangle, where a ray intersects a triangle if the equation $\beta\overrightarrow{(B-A)} + \gamma\overrightarrow{(C-A)} - t\vec{u} = \overrightarrow{(O-A)}$ has a positive solution with $\beta \in [0,1]$ and $\gamma \in [0,1]$. However, since a mesh consists of a large number of triangles, such process is extremely time consuming. A possible way to speed up is to define a box shape with the function `bound`, so that every vertex of the mesh belongs to its volume. We can then begin the testing for intersection with the mesh only if there is an intersection with its bounding box. For even more optimization, we can further divide the box and split the triangles into sets according to the box they belong to with the function `bvh`. This allows us to reduce the amount of triangles tested for intersection, thus reducing the rendering time, as can be seen in Figure 5.



(a) No acceleration structures, 32 samples per pixel rendering time - 1705 seconds

(b) Single bounding box, 32 samples per pixel rendering time - 359 seconds

(c) BVH, 32 samples per pixel rendering time - 59 seconds

Figure 5: Mesh rendering with 3954 triangles

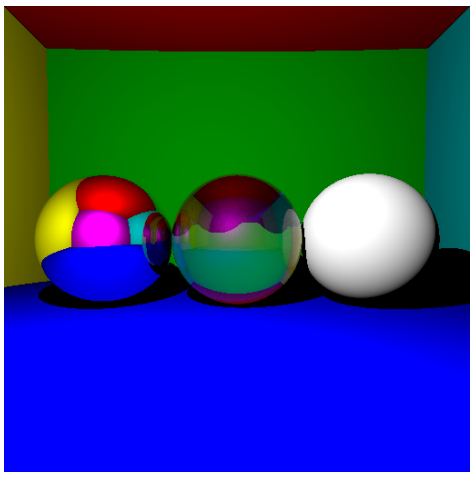
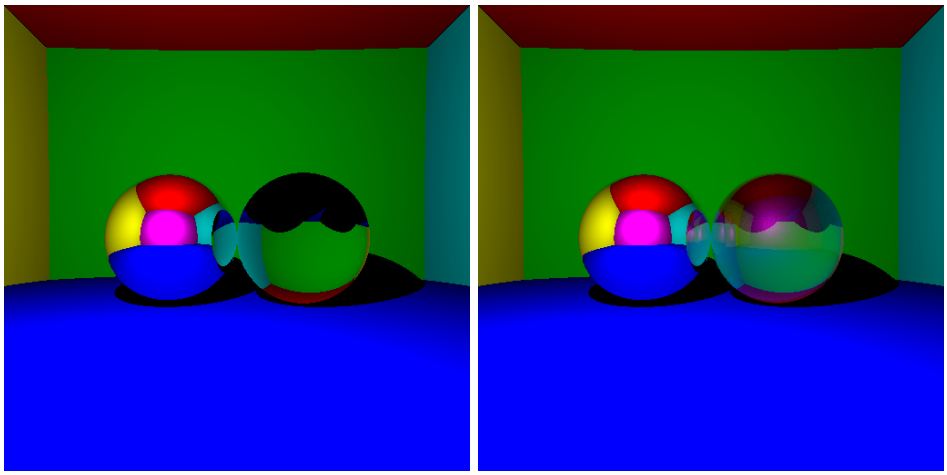


Figure 6: Types of surfaces

Surface types The behavior of a ray upon intersection with a object depends on its surface. The geometry class supports three types of surfaces as shown in Figure 6 - mirror (left), transparent (center) and diffusive (right).
In case of a mirror surface, the ray is reflected around the normal of the surface.
In case of a transparent surface, the ray is refracted according to the Snell-Descartes law. We furthermore apply the Fresnel law, thus reflecting a portion of the light. The difference adding Fresnel makes can be seen in Figure 7.

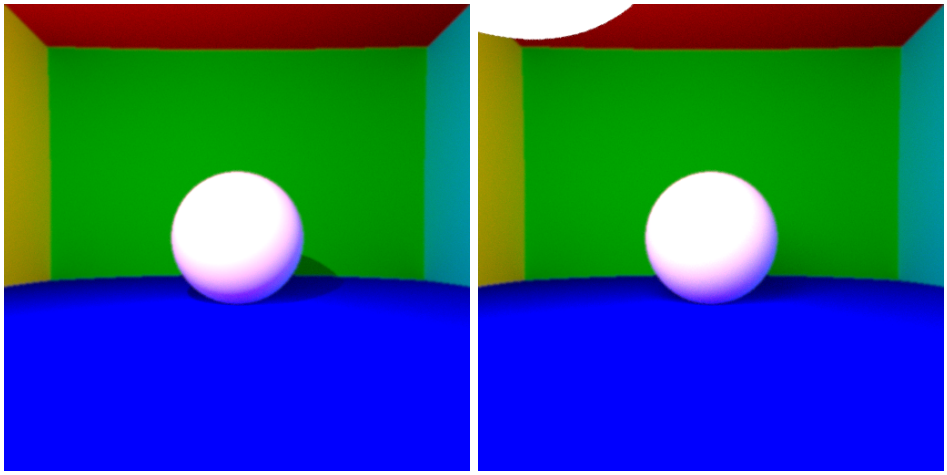


(a) without Fresnel law (b) with Fresnel law

Figure 7: Reflection and refraction

If the ray has reached a diffusive surface we proceed to compute its color.
In order to ensure that the program terminates, we restrict the number of ray bounces to five.

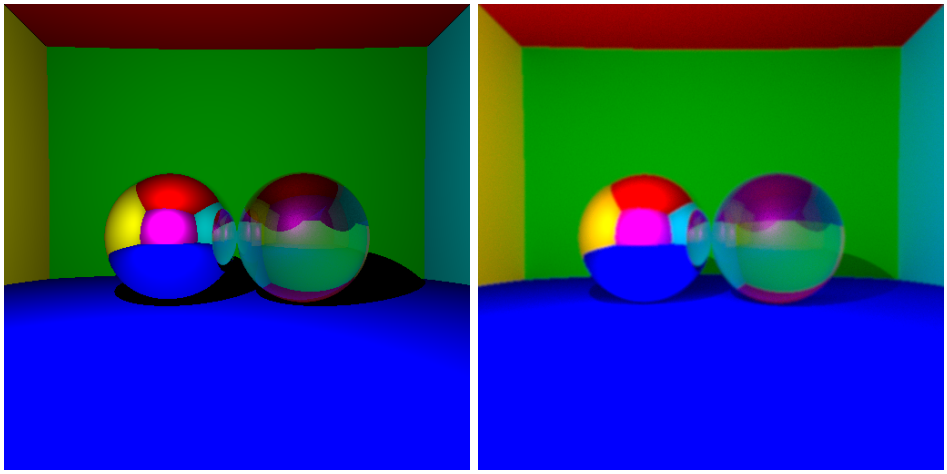
Light source To realistically compute the color, we must evaluate the contribution from the light source. We are going to use light with center $S = (-20, -10, 40)$ and intensity $I = 2 * 10^{10}$. As presented in Figure 8, the code supports two types of light sources, namely point and area light sources.



(a) point light (b) area light

Figure 8

A naive approach is to use the formula $\frac{I}{4\pi||S-P||^2}\frac{\rho}{\pi}V < N, \frac{S-P}{||S-P||} >$, where P is the intersection point, N is the normal to the surface at point P, ρ is the color and V indicates whether the light source is directly visible.
However, for a more natural look, we add indirect lighting using the Monte Carlo algorithm. The difference it makes is showcased in Figure 9



(a) only direct lighting (b) direct and indirect lighting

Figure 9

Finally, to account for the fact that computer screens do not react linearly with pixel intensities, we add gamma correction with $\gamma = 2.2$.

Full record of all obtained images and their rendering times can be found in `/_results/performance.txt`

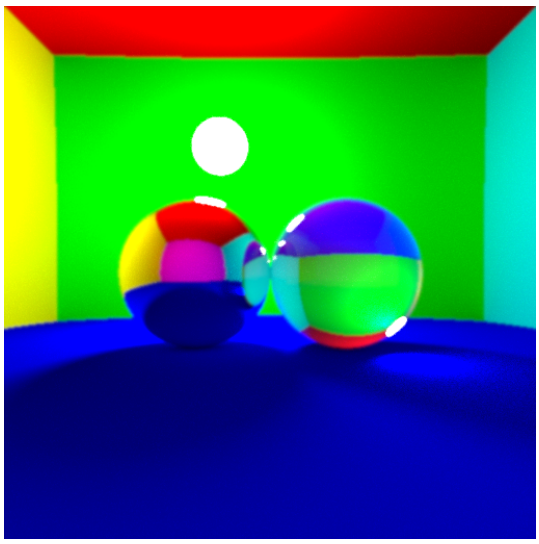


Figure 10: Another example with 5000 samples per pixel and rendering time 1404 seconds