Project on Ray Tracing

Université de Montpellier, M1 IMAGINE

Adèle Imparato

December 2022

Contents

1 Introduction

The goal of this project is to learn how to trace rays of light into a scene made of 3D objects. These rays of light allow for a more realistic rendering. This project is split into four phases. The first one consists of tracing rays into a scene such that the rendering shows the shapes in the scene in their original color. The second phase uses what was done in phase 1 and adds light reflection and shadows. Then, the third phase consists of adding a 3D mesh object into the scene and render it in a similar way. The final phase specifies the material of the spheres (invisible, mirror, glass) and aims to reduce the computational time by implementing a KdTree or simply using bounding boxes.

2 First phase

This first phase teaches us the basics of ray tracing by evaluating the color of a fragment thanks to rays of light pointing to the objects of the 3D scene.

2.1 Implementation of *rayTrace()*

Once the project is launched, the user is allowed to press on the *r* key. This key enables to trace a set of rays calling the function *ray_trace_from_camera()*. This function aims to fill the values of the colors of each pixel of the image such that *rendu* becomes the output of the scene once illuminated. As it loops through all pixels of the image, it therefore traces 230400 rays (480x480, the size of the image). Moreover, each pixel is made of several samples defined by the variable *nsamples*. For each sample, the function *rayTrace()* is called.

rayTrace() aims to compute all intersections with all geometric objects of the scene for a given ray. *NRemainingBounces* specifies the depth of the tree (used in Phase 3 for reflection). To compute the intersections, this method calls *rayTraceRecursive()* that is generating the tree of rays. It is important to note the use of the variable *znear* that allows to ignore the intersections that are before that value. It is especially useful when visualising the Cornell box. In fact, since it is a box, we do not want to consider the intersections of the rays with the front wall as it would hide the inside of the box. We therefore compute the initial value for *znear* by dividing a value of 4.15 (chosen after many trials) by the dot product of the direction of the camera and the direction of the ray. This allows to see inside the box just as shown by the scene (no matter its orientation).

rayTraceRecursive() takes a ray as parameter and compute its intersection in the scene (see section 2.4) instantiating the variable *raySceneIntersection*. If the intersection exists, it means the ray has touched and object, therefore the function calls *phongLight()* and *softShadow()* (see section 3) in order to return the color value at that intersection point. If it deals with a sphere of material type *Material_Mirror*, it uses recursion to compute the reflecting rays of light (note that the *znear* must be smaller as we do not need to deal with the front

wall anymore). If there is not intersection at all, the function returns the color black.

2.2 Implementation of *Intersect()* in Sphere.h

In order to compute the intersection between a ray and a sphere, we use the function *intersect()* of the class *Sphere.h*. This function aims to find the roots (t_1 and t_2) of this specific equation:

$$t = \frac{-2 \cdot < d, CO > \pm \sqrt{2 \cdot < d, CO >^2 - (4 \cdot < d, d > \cdot ||CO||^2 - r^2)}}{(2 \cdot < d, d >)}$$

with r coding for the radius of the sphere, CO coding for the distance between the center of the sphere and the origin of the ray and d the direction of the ray.

These two roots represent the distances from the origin of the ray until the intersection. It is crucial to only consider the distances that are positive (meaning that the intersection is not behind the camera) and eventually to select the closest intersection. Once the value is selected, the function simply does $ray.o + t * ray.d$ in order to get the intersection point as a Vec3 object. Lastly, the function returns an object *RaySphereIntersection intersection* containing all the necessary data of that intersection. If it could not find a proper intersection point, it sets the field *intersection.intersectionExists* to false.

2.3 Implementation of *intersect()* in Square.h

For the square intersection, the principle differs a bit. The easiest way to obtain the intersection between a ray and a quad is to first compute the intersection between the ray and the plan sketched by the front face of the quad. To do so, we need to solve this equation:

$$t = \frac{< bottomLeft(), normal() > - < o, normal() >}{< d, normal() >}$$

with d the direction of the ray, o the origin of the ray, *normal()* the normal of the quad and *bottomLeft()* the bottom left corner of the front face of the quad.

Once the intersection between the ray and the plan is found, we verify whether the point of intersection is inside the quad. To do so, it is necessary to re-propose a new x-axis and a new y-axis. The new x-axis corresponds to *bottomRight() - bottomLeft()* and the new y-axis to *upLeft() - bottomLeft()*. Then, the point of intersection is projected on these new axes allowing to compute u and v. If $u \geq 0$, $u \leq$ the norm of the new x-axis, $v \geq 0$ and $v \leq$ the norm of the new y-axis, then it means the intersection is in the quad. The function then returns an object *RaySquareIntersection intersection* filled with the necessary data. If the intersection is outside the quad, the function sets *intersection.intersectionExists* to false and returns the object *intersection*.

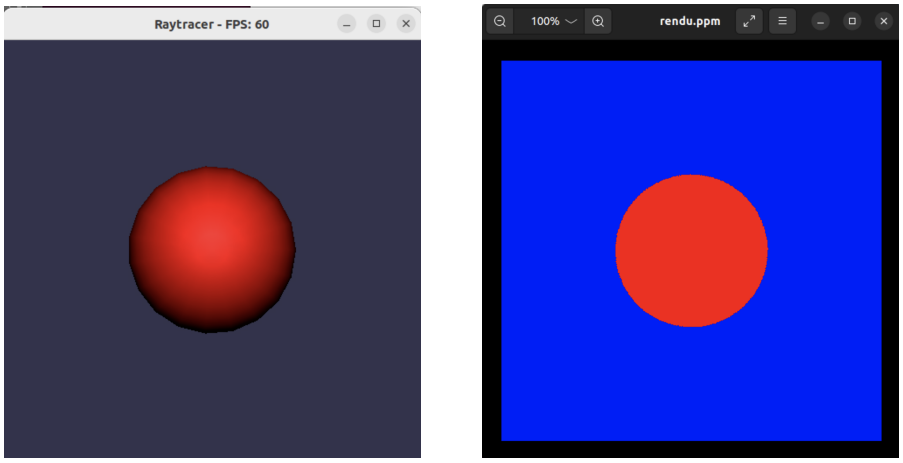
2.4 Implementation of *computeIntersection()*

In order to compute the intersections of the rays with the geometric objects, *rayTraceRecursive()* calls the function *computeIntersection()* for each ray. This function's role is to check among the list of spheres, squares and meshes whether the ray is going to touch the geometric shape at some point. It then returns an object *RaySceneIntersection result* corresponding to either *RaySphereIntersection*, *RaySquareIntersection* or *RayTriangleIntersection* depending on the geometric object that was intersected. For each ray, we only consider the closest intersection point.

2.5 Results

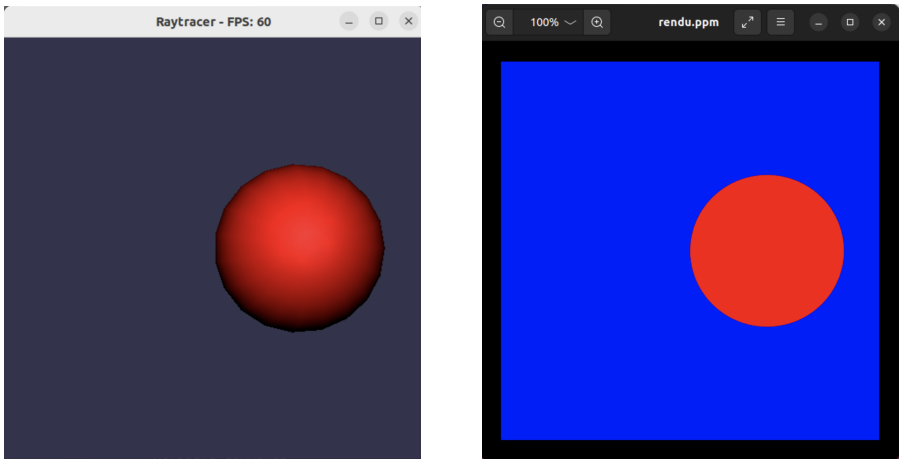
To verify our functions, several tests have been conducted.

2.5.1 Scene 0: Single sphere



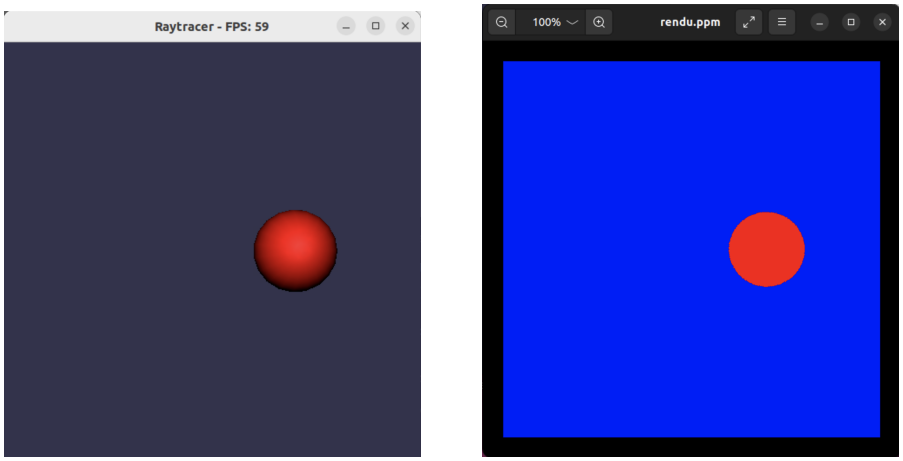
(a) (b)

Figure 1: Initial scene 0.



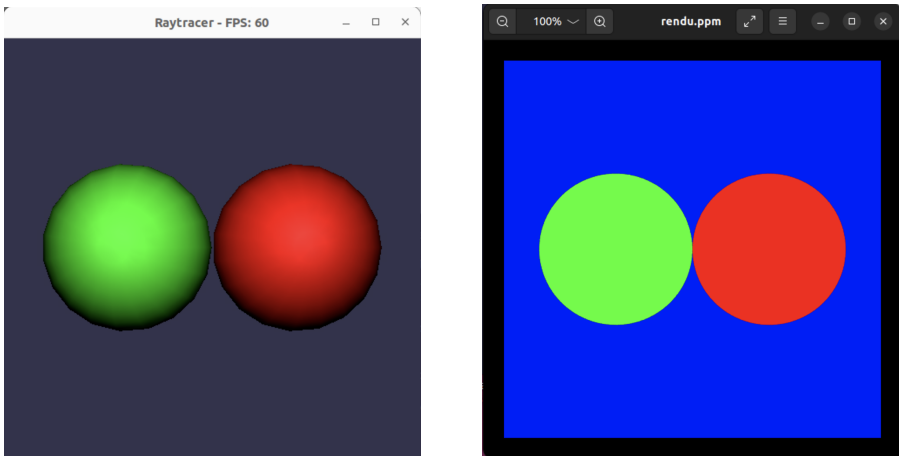
(a) (b)

Figure 2: Scene 0 after setting the center of the sphere to Vec3(1.0, 0.0, 0.0)



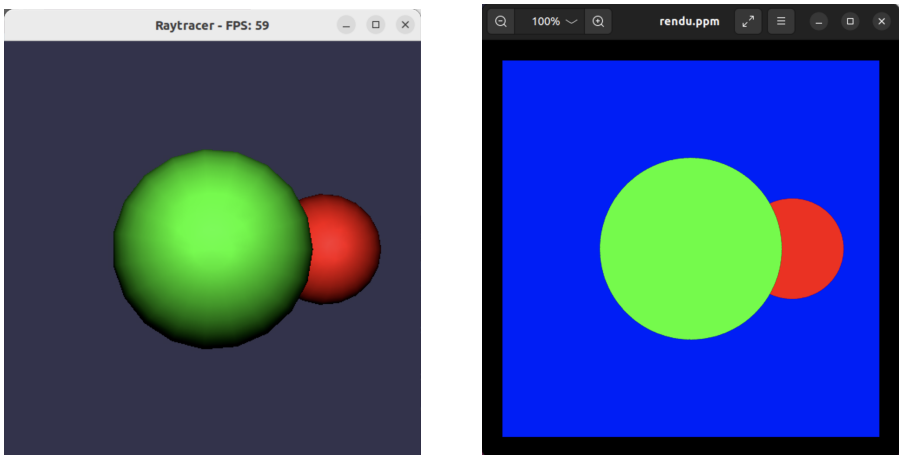
(a) (b)

Figure 3: Scene 0 after setting the center of the sphere to Vec3(1.0, 0.0, 0.0) and setting the radius to 0.5.



(a) (b)

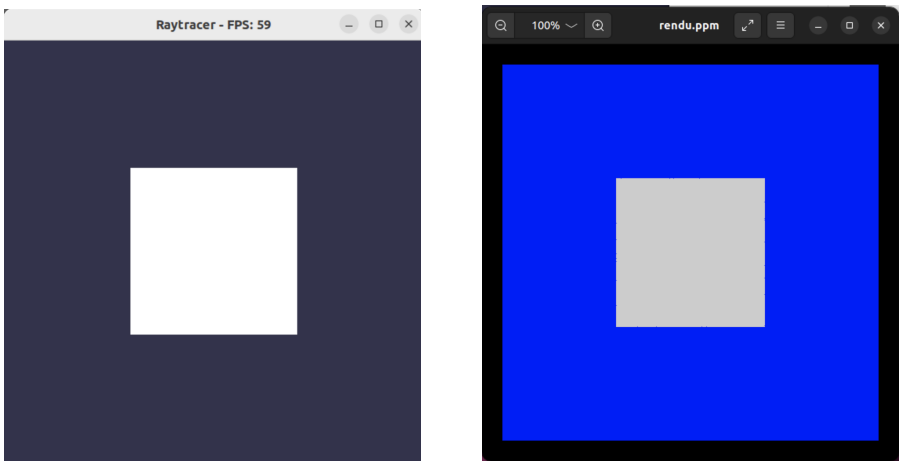
Figure 4: Scene 0 after adding a green sphere and editing the centers.



(a) (b)

Figure 5: Scene 0 after adding a green sphere and putting it ahead of the red sphere. The center of red sphere is set to Vec3(2., 0., -3.) and the green sphere to Vec3(0., 0., 1.).

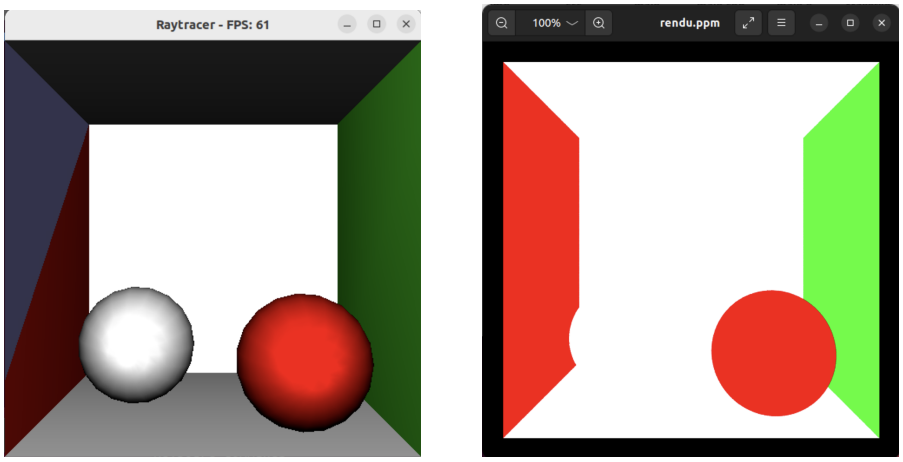
2.5.2 Scene 1: Single square



(a) (b)

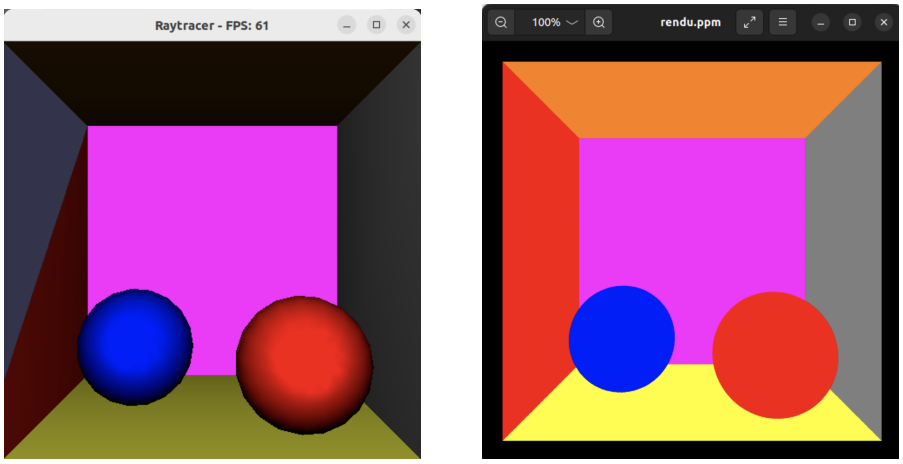
Figure 6: Initial scene 1.

2.5.3 Scene 2: Cornell box



(a) (b)

Figure 7: Initial scene 2.



(a) (b)

Figure 8: Scene 2 after editing the colors.

3 Second phase

In this second phase, the goal is to make the scene more realistic by adding shadows and light reflection.

3.1 Phong's reflection model

Phong's reflection model is a way to compute the color of a geometric shape once illuminated by evaluating the three different kind of lights at each position on the object.

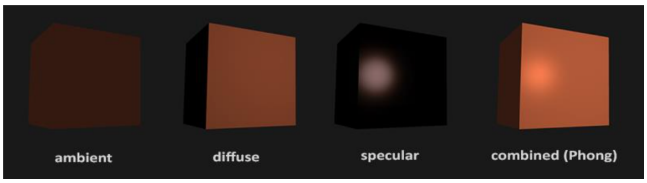


Figure 9: The three types of light reflections used to enhance a 3D scene. Their combination is the Phong's model.

This model is implemented through the function *phongLight()*, which is called in *rayTraceRecursive()*. The function simply makes use of the following formula:

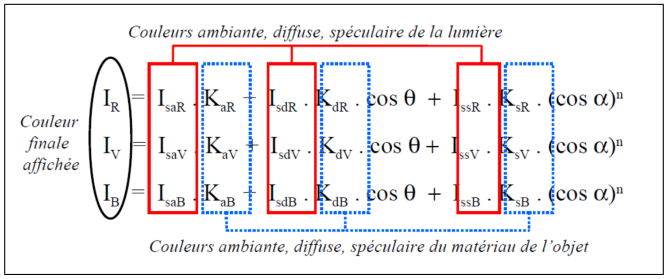


Figure 10: Formula used to compute the three color components computed using Phong's model.

with V, R, L, theta and alpha being represented in the following schema:

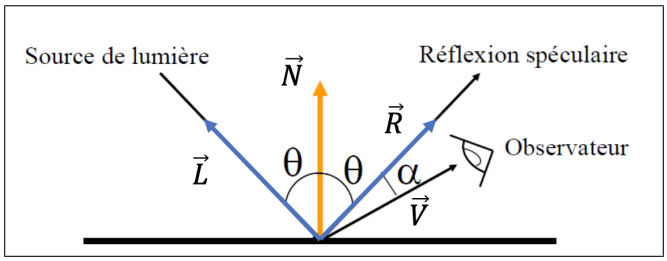
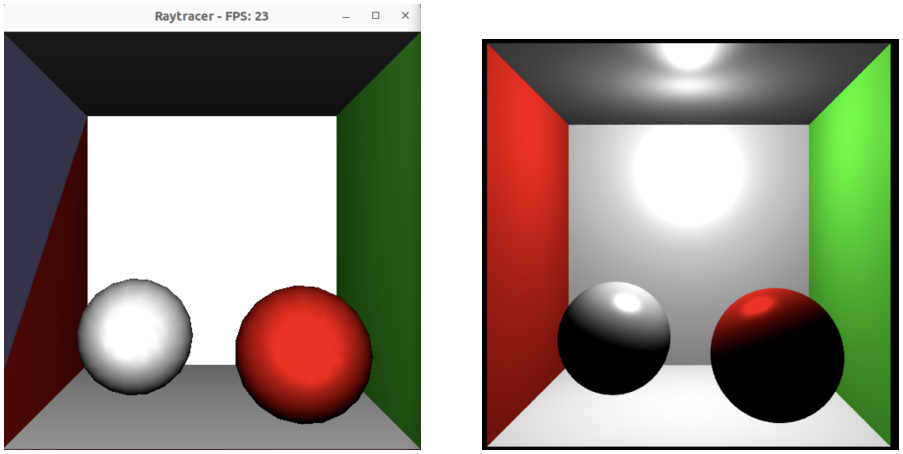


Figure 11: How is light perceived by the eye considering a source of light illuminating a geometric object.

3.1.1 Result



(a) (b)

Figure 12: Initial scene 2 after implementing Phong's reflection model.

3.2 Shadows

To make this scene even more realistic, it is necessary to consider the shadow areas underneath the objects. To do so, for each intersection point, we trace a ray from that point to the source of light. At first, if this ray is intersected by any geometric shape, its color gets switched to black so that it darkens the area that does not receive light just as shown in the following schema. This method is called *isShadow()*.

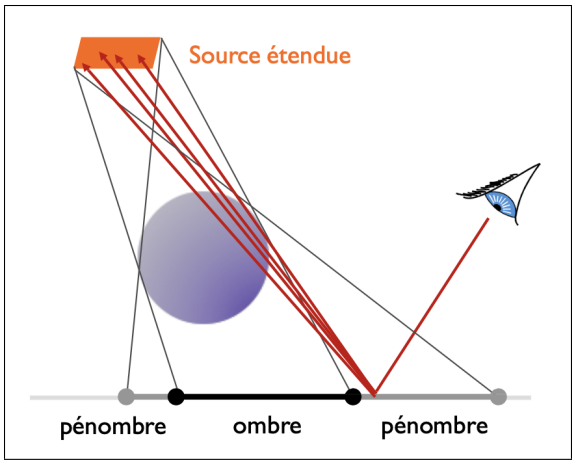
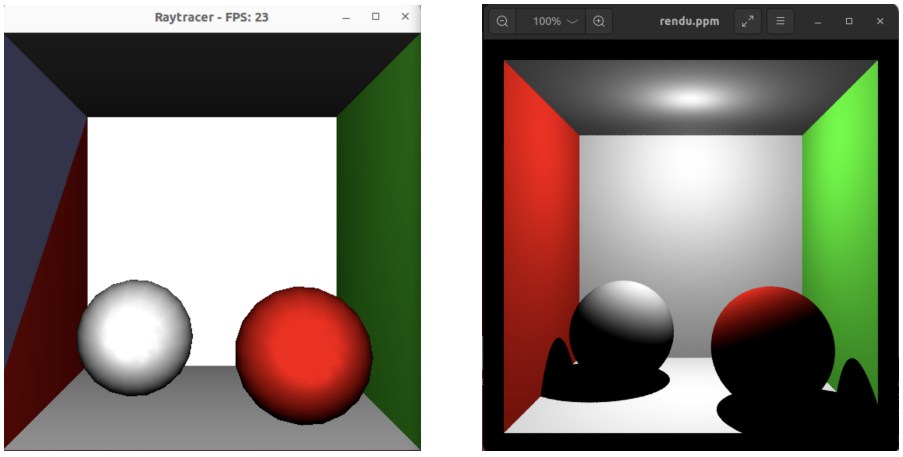


Figure 13: Where is shadow located when an object is illuminated by a source of light.

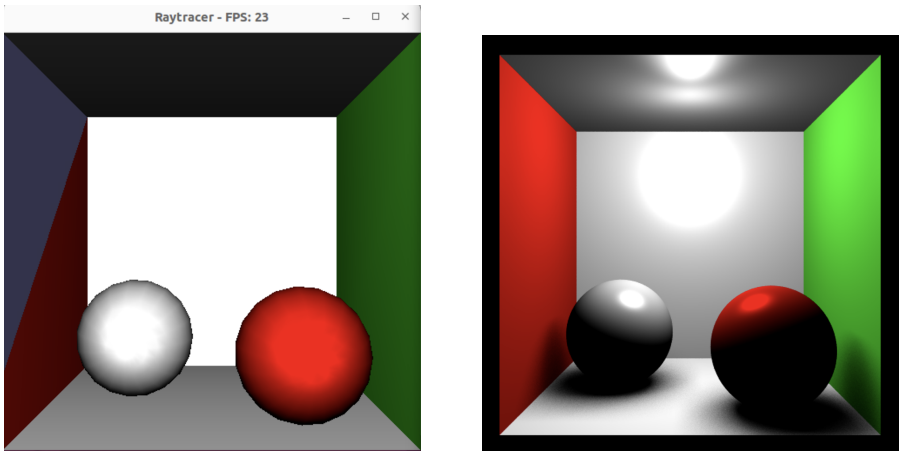
Secondly, it is interesting to consider the areas denoted as 'pénombre' in the schema as they are areas of soft shadow. This type of shadow gives an even a better outcome as it enables the shadows to fade. We therefore create a method called *softShadow()* that modifies the color returned by *phongLight()* by darkening it depending on the amount of rays able to reach that point. The function works as follow: it generates a set of 20 rays around the x and z axis of the initial light position meaning that instead of throwing one ray for one intersection, we will throw 20 rays for one intersection. For each of these 20 rays, we check whether it intersects with an object (sphere, square or mesh). Eventually, we multiply the initial color (computed with Phong) by the percentage of rays that were not intersected. Moreover, in this function, we consider an *offset* variable that is set to 0.0001, this value enables to avoid considering intersections as inside the object when it is in fact outside. Eventually, the variable *ignore* enables to ignore the small values of t.

3.2.1 Result



(a) (b)

Figure 14: Initial scene 2 after adding shadow.



(a) (b)

Figure 15: Initial scene 2 after adding soft shadows.

4 Third phase

In this third phase, we would like to be able to add a 3D mesh into scenes such that it gets rendered the same way as spheres and squares, that is, each of its triangles being correctly intersected.

4.1 Adding a 3D mesh into a scene

Adding a 3D mesh object into a scene is rather simple. We simply need to load an .off file and make it a Mesh object. We can for instance add our friend Suzanne inside our Cornell box like this:

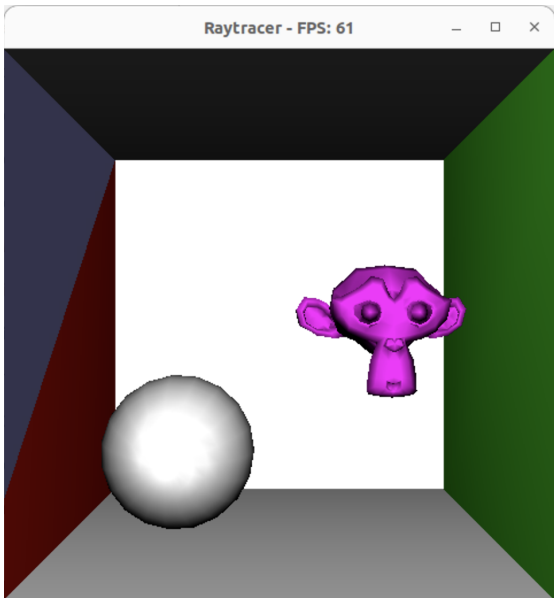


Figure 16: Adding a Mesh object (suzanne.off) into the initial scene 2.

4.2 Light intersection with 3D Mesh

To allow meshes being illuminated, we need to compute the intersection for each triangle of the mesh object with the ray of light. To do so, we write the function *intersect()* of the class *Mesh.h* so that it goes through all triangles of the mesh object and for each one of them, it computes its intersection with light. Eventually, only the closest intersection is kept. Let's now focus on how the method *intersect()* of the class *Triangle.h* works. This function verifies three conditions in order to decide whether there is an intersection between this triangle and the ray passed by parameter. It eventually returns an object *RayTriangleIntersection* that is filled with the correct data. The first condition verifies whether the ray is parallel to the triangle object. If yes, there won't be any intersection. Otherwise it aims to find t, the distance from the origin of the ray to the intersection point with the plane generated by the triangle, as follow:

$$t = \frac{(a - p) \cdot n}{d \cdot n}$$

with a being a point on the plane, p a point on the ray, d the direction of the ray and n the normal of the plane.

Then, the functions checks whether t is greater than 0.00001 to make sure it is in the same direction as the ray and not behind the camera. If this condition is true, it then computes the intersection point p with the plane and aims to verify whether that point belongs to the triangle area. To do so, we compute the barycentric coordinates ($w0$, $w1$, $w2$) and check that they are ≥ 0.0 and ≤ 1.0 as well as if the normals of the sub-triangles (sketched in light blue on the following schema) are pointing in the same direction than the normal of the triangle. If this is the case, than the intersection point p in within the triangle.

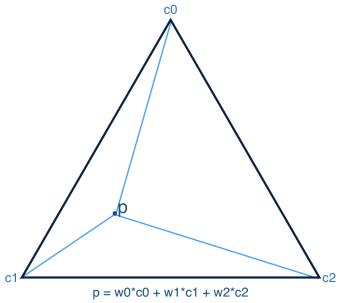
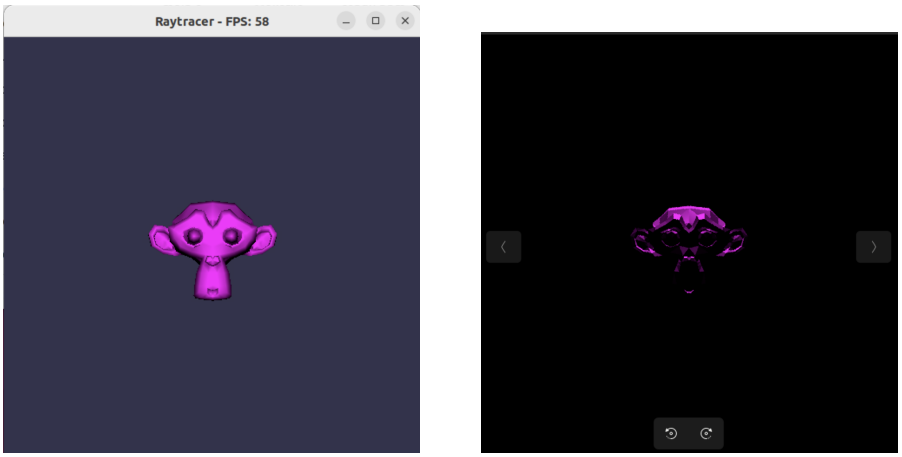


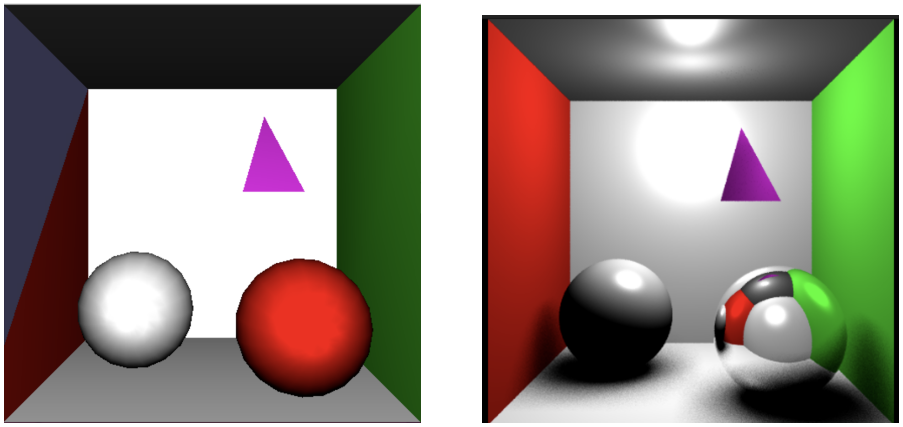
Figure 17: Representation of an intersection point inside a triangle and link with barycentric coordinates.

4.2.1 Results



(a) (b)

Figure 18: Scene made of a single Suzanne showing how ray intersects a 3D mesh.



(a) (b)

Figure 19: A single triangle mesh added into the Cornell box.

5 Final phase

5.1 Adding a mirror sphere

Surprise ! The red sphere is not red, in reality it is a mirror, meaning that it should reflect all objects around it. To do that, we include a condition that considers the material type *Material_Mirror* so that it recursively calls the function *rayTraceRecursive()* giving it as arguments a new ray, a small *znear* and *NRemainingBounces*-1. The new ray is computed following the physics of reflection.

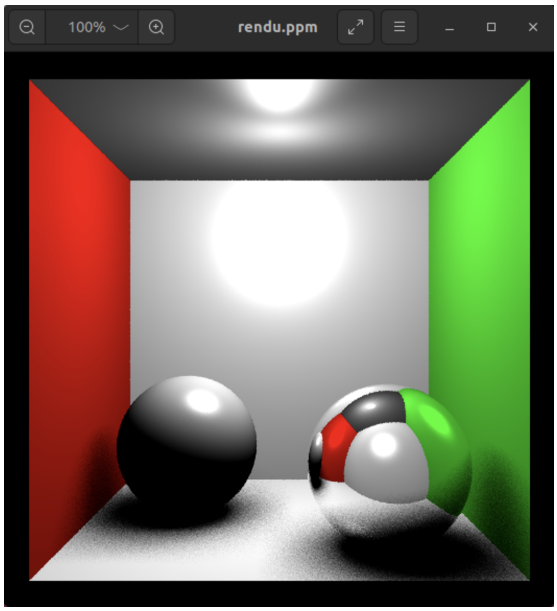


Figure 20: Initial scene 2 with a mirror sphere.

5.2 Adding a transparent sphere

We then aim to add a glass sphere (*Material_Invisible*[1]) that is transparent. To do so, we need to consider the index of the medium, that is 1.4 in a glass sphere and 1.00029 in the air (1.00029 may be rounded to 1.0). The fact that the air and the sphere have a different index influences the way the ray is propagated inside the sphere. On the following schema, it is shown how the ray gets refracted:

[1]I have created a material Invisible for that transparent sphere and used *Material_Glass* for another type of refraction in the Section Extra (Transparent Marble).

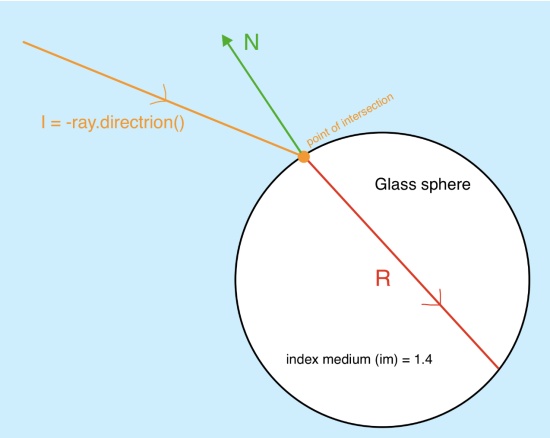


Figure 21: Schema of a refracted ray inside the glass sphere.

The direction of the refracted ray is computed as follow:

$$R = im \cdot (-I) + im \cdot (dot(-I, N) + \sqrt{im^2 \cdot (dot(-I, N))^2}) \cdot N$$

The refracted ray is then created with a direction R and an origin point corresponding to the point of intersection between the initial ray and the sphere (Figure 21 in orange) $+\epsilon \cdot R$ to avoid the issue of self intersection.

Just as for the mirror sphere, we use recursion so that the intersection eventually reaches a colored point and can return that value back in the recursion.

In order to get an even more realistic rendering, we soften the shadow below a glass sphere so that it looks transparent. Here is the result:

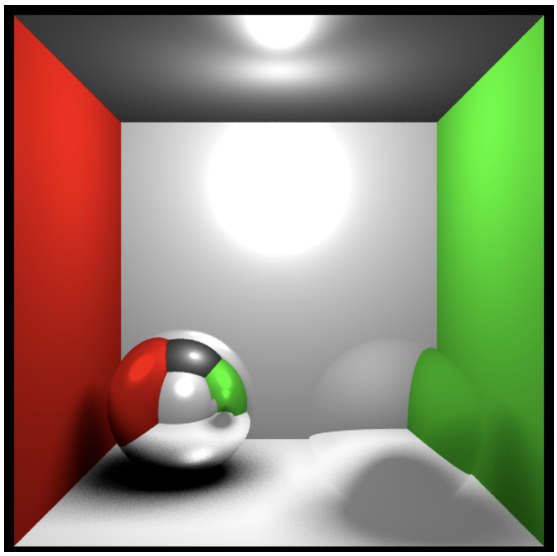


Figure 22: Initial scene 2 with a mirror sphere and a glass sphere.

6 Extra

6.1 Implementation of a KdTree

A kdTree is a data structure able to partition space hierarchically into regions so that it can be easily visited. In the context of this project, a kdTree might be really useful in order to reduce the computational time, particularly when it comes to compute the intersection with a mesh. In fact, since a mesh is made of a large amount of triangles, computing the intersection between a ray and the mesh object requires to compute the intersection with all triangles of the mesh each time. To improve the computational time, we implement a kdTree on mesh objects. Below, the pseudo code to build the tree on meshes:

```
2   struct TreeNode {
3       node_data_triangle;
4       TreeNode leftChild;
5       TreeNode rightChild;
6       isRoot;
7       isLeaf;
8       depth;
9       bounding_box;
10  };
```

Figure 23: Pseudo-code of the structure TreeNode used to build a KdTree.

```
buildKdTree(){
    create TreeNode scene_tree;
    scene_tree.isRoot = true;
    scene_tree.depth = 0;
    build its bounding box bb;
    scene_tree.bounding_box = bb;

    for all meshes of scene{
        for all triangles of mesh{
            scene_tree.node_data_triangle.add(triangle);
        }
    }
    min_objetcs = 10;
    call recursiveKdTree(t, scene_tree.depth++, min_objects);
}
```

Figure 24: Pseudo-code to start the process of building a KdTree.

```
recursiveKdTree(parent_node, current_depth, min_objects){
    split largest dimension of the bouding box in two

    create TreeNode child1, child2;
    child1.depth = current_depth;
    child2.depth = current_depth;

    create and set bounding boxes for child1 and child2

    for all triangles in parent_node{
        if triangle is in child1.bounding_box{
            child1.node_data_triangle.add(triangle)
        }
        if triangle is in child2.bounding_box{
            child2.node_data_triangle.add(triangle)
        }
    }

    parent.leftChild = child1;
    parent.rightChild = child2;

    if(child1.node_data_triangle.size > min_objects) {
        recursiveKdTree(child1, current_depth++, min_objects);
    }else{
        child1.isLeaf = true;
    }

    if(child2.node_data_triangle.size > min_objects) {
        recursiveKdTree(child2, current_depth++, min_objects);
    }else{
        child2.isLeaf = true;
    }
}
```

Figure 25: Pseudo-code to recursively build a KdTree.

Here is what it looks like once the KdTree gets drawn on the scene for different values of *min_objects* (the maximum amount of triangles allowed in one leaf).

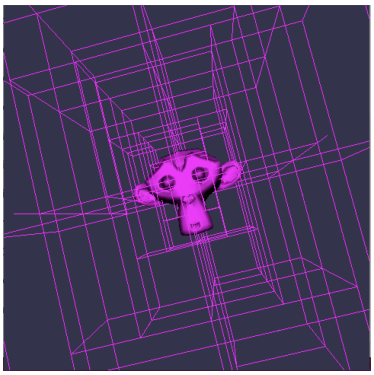


Figure 26: KdTree around Suzanne when *min_objects* = 100.

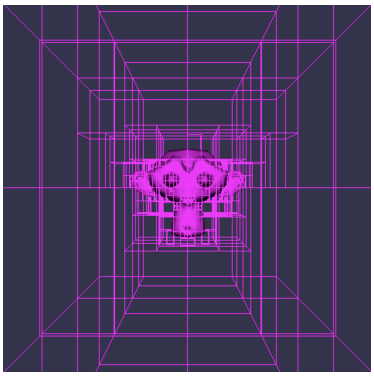


Figure 27: KdTree around Suzanne when *min_objects* = 10.

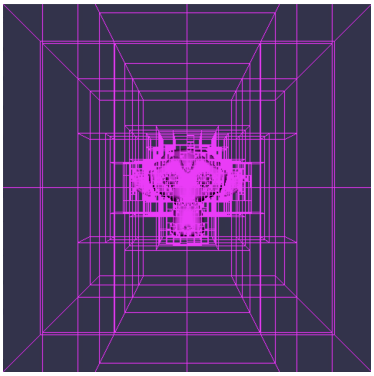


Figure 28: KdTree around Suzanne when *min_objects* = 1.

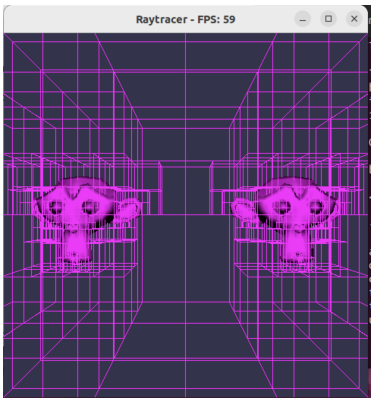
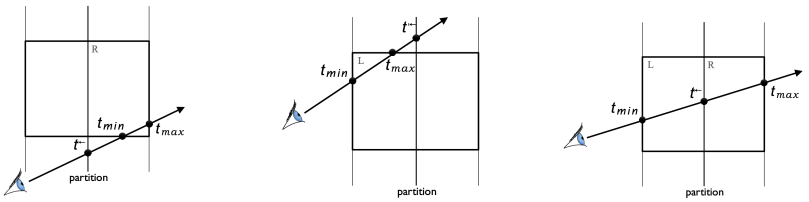


Figure 29: KdTree around two mesh objects (Suzanne) when $min\_objects = 10$.

Once the tree is build, we need to visit it such that each time we trace a ray, it computes the intersections only with the triangles of the closest leaf it intersects with. To do so, here is the pseudo-code of how it is supposed to work (SPOILER: I wasn't able to make it fully work):

```
63  RayTriangleIntersection recursiveVisitKdTree(ray, current_node, t_min, t_max){
64
65      if(current_node.isLeaf){
66
67          find closest intersection among triangles in current_node
68          return closest intersection;
69      }
70
71      t = intersectBox(ray,current_node.leftChild.bounding_box);
72
73      if (t <= t_max){ // CASE ONE traverse second box only
74          return recursiveVisitKdTree(ray, current_node.rightChild, t_min, t_max);
75      }else if (t >= t_max){ // CASE TWO traverse first box only
76          return recursiveVisitKdTree(ray, current_node.leftChild, t_min, t_max);
77      }else { // CASE THREE traverse both boxes
78          t_hit = recursiveVisitKdTree(ray, current_node.leftChild, t_start, t);
79          if(t_hit <= t){
80              return rti_hit; // early termination
81          }
82          return recursiveVisitKdTree(ray, current_node.rightChild, t, t_max);
83      }
84  }
```

Figure 30: Pseudo-code to visit the KdTree recursively. [2]



(a) Case 1 (b) Case 2 (c) Case 3

Figure 31: Three different cases occurring when visiting the KdTree.

6.2 Alternative to KdTree

As I didn't manage to debug my KdTree, another option might be used (not optimal though). This alternative consists of using a bounding box around a mesh, so that we only compute the intersection with its triangles if the ray intersects with that box. Here is how the bounding box is built around a mesh (Here, Suzanne).

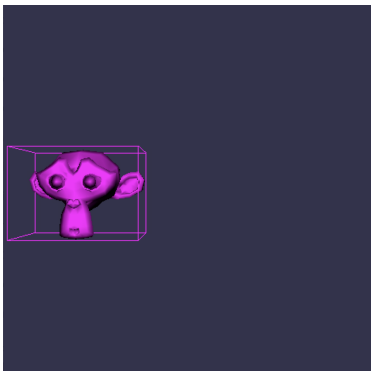


Figure 32: Bounding box built around a single mesh (suzanne.off).

This method considerably reduces the computational time as it takes about 1.5 minute to trace rays on that scene with a single Suzanne without bounding box, and only about 15 seconds with the use of a bounding box.

6.3 Transparent marble

While implementing the transparent sphere (*Material_Invisible*) I realised there was another way of refracting the ray, giving a different kind of result, similar to a transparent marble. I therefore used the material *Material_Glass* for that second way of refracting a ray. Here is how the refracted ray is computed based on Figure 21 and Snell's Law [1]:

$$R = (n_r(N \cdot I) - \sqrt{1 - n_r^2(1 - (N \cdot I)^2)})N - n_r I$$

$$n_r = \frac{1.00029}{im}$$

Here is what it looks like:

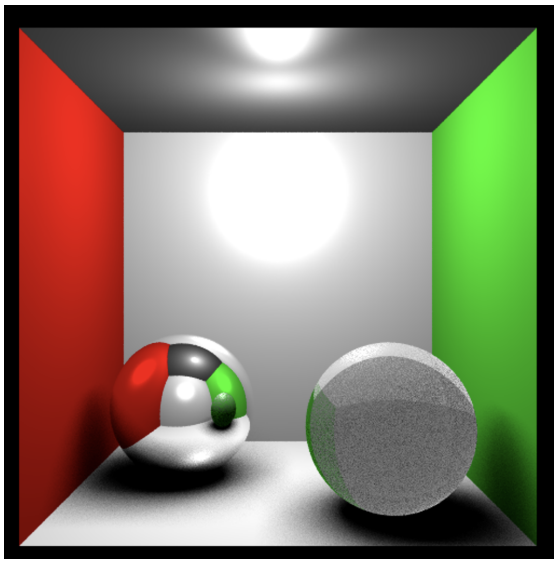
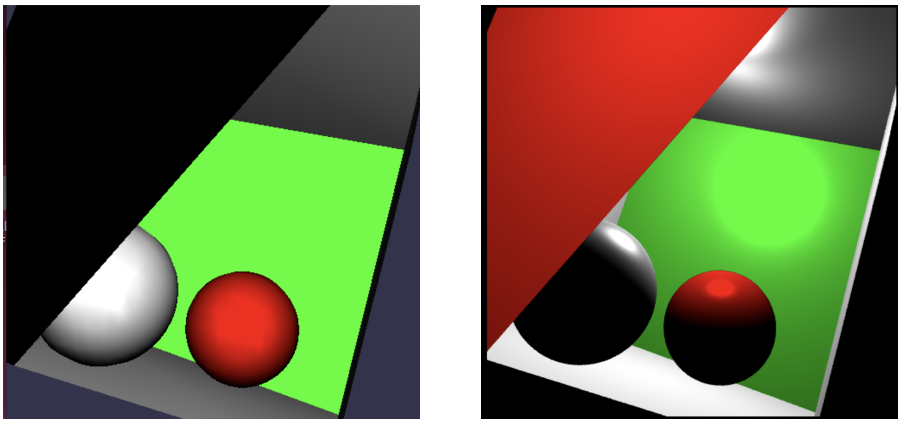


Figure 33: Initial scene 2 with a noisy transparent marble.

This result is quite close to what we could expect, except is has some noise which I didn't managed to get rid of. Moreover, it would have been nice adding some extra reflection to render the transparent marble even more realistic.

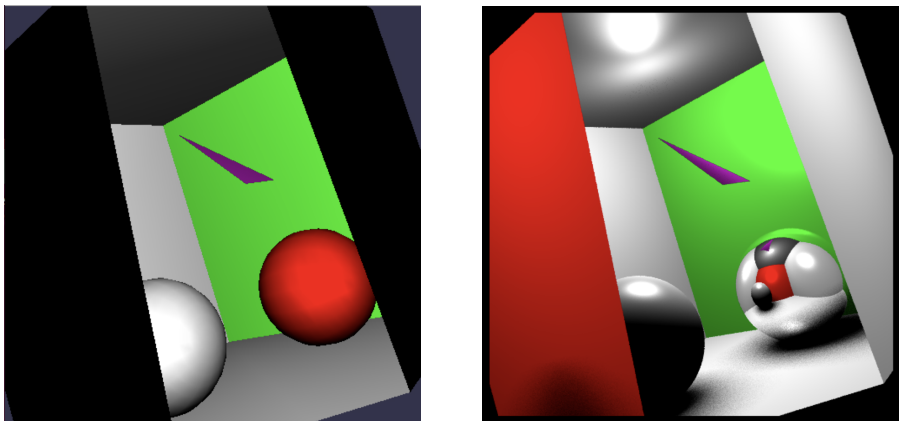
7 Tests

In this section, we aim to prove how strong our code is by tracing rays in various scenes, using different orientations and objects.



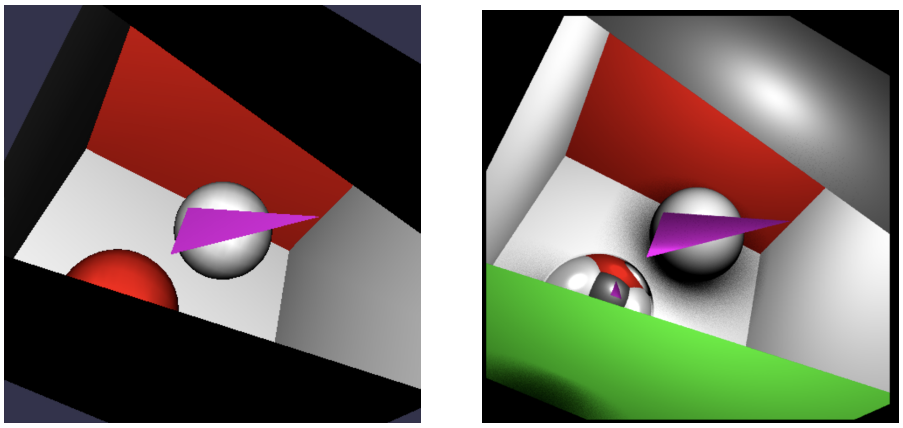
(a) (b)

Figure 34: Cornell box, test 1.



(a) (b)

Figure 35: Cornell box, test 2.

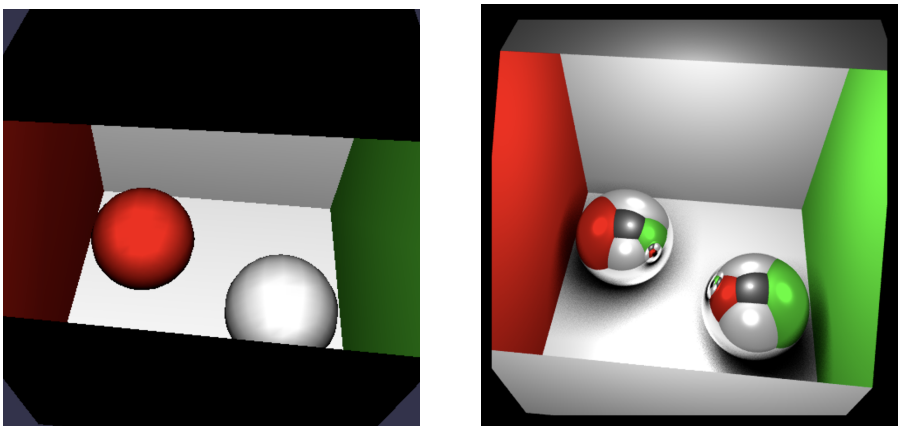


(a) (b)

Figure 36: Cornell box, test 3.

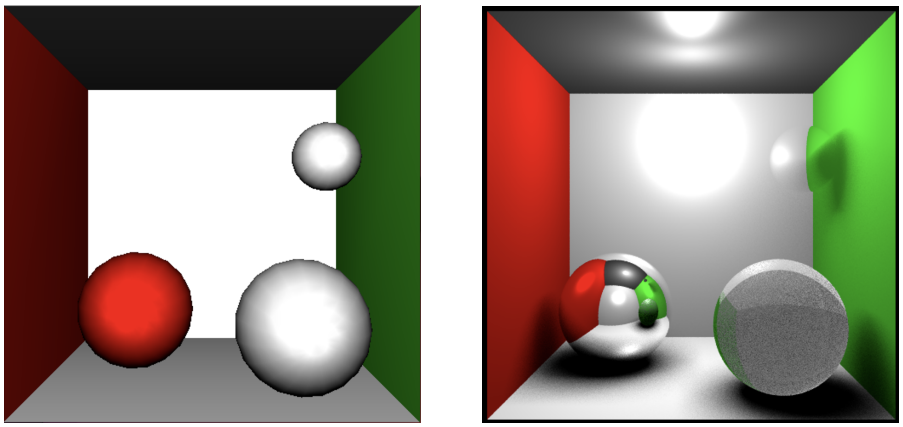
(a) (b)

Figure 37: Suzanne, test 1.



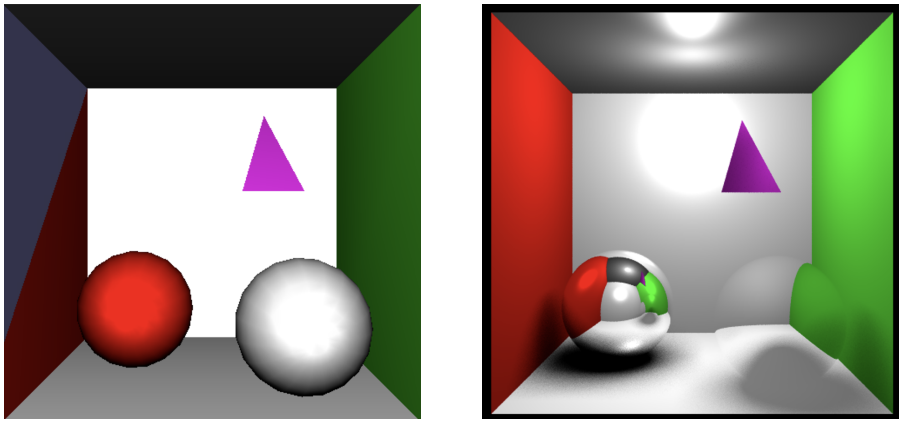
(a) (b)

Figure 38: Cornell box, test 4.



(a) (b)

Figure 39: Cornell box, test 5.



(a) (b)

Figure 40: Cornell box, test 6.

References

[1] Justin Solomon. *Introduction to Computer Graphics (Lecture 11): Ray tracing; reflection and refraction; ray trees*. Dec. 2020. URL: https://www.youtube.com/watch?v=Tyg02tN9oSo.

[2] Justin Solomon. *Introduction to Computer Graphics (Lecture 12): Accelerating ray tracing; bounding volumes, Kd trees*. Dec. 2020. URL: https://www.youtube.com/watch?v=TrqK-atFfWY.