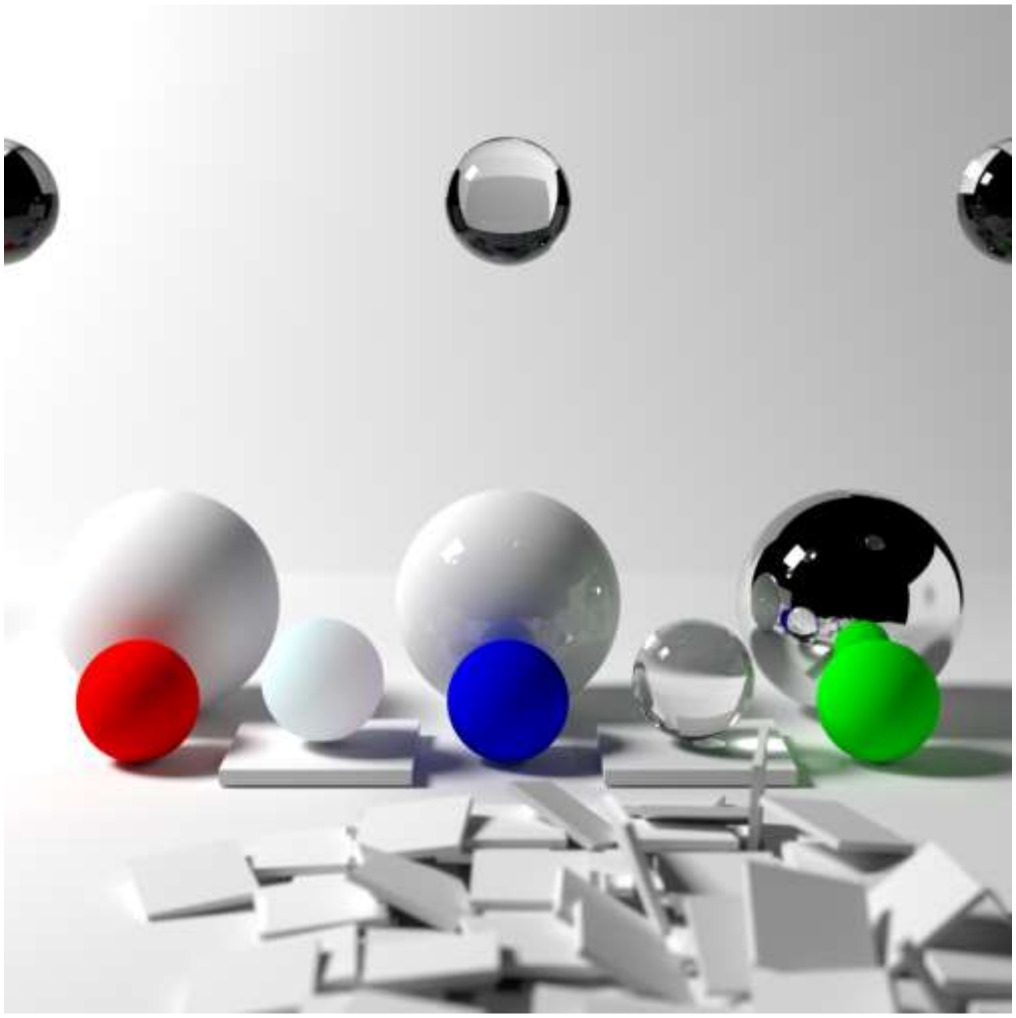
BIDIRECTIONAL PATH TRACING



ÍÑIGO FERNÁNDEZ ARENAS

CS500

INDEX

1. INTRODUCTION

Both the eye point of the viewer and the set of primary light sources in a scene have always been identified as being important for solving the global illumination problem to create a realistic rendering. Some algorithms such as ray tracing are entirely built around the importance of the viewing point. Other algorithms such as the progressive radiosity method put a great emphasis on the contributions of the light sources. Ideally one would want an algorithm which considers the importance of both the light sources and the viewing point. In this report we tried to implement an algorithm which treats light sources and the viewing point on an equal basis.

1.1 PREVIOUS WORK

The implementation is based on a previous implementation of a **traditional path tracer**, in which the idea of the algorithm is to throw a ray for each pixel on the screen and bounce that ray inside the scene accumulating all the flux of each pixel until we reach the light. We know that the radiance that reaches a surface in a scene is defined as follow:

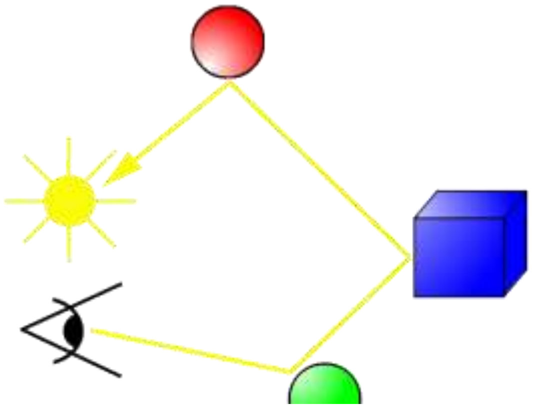
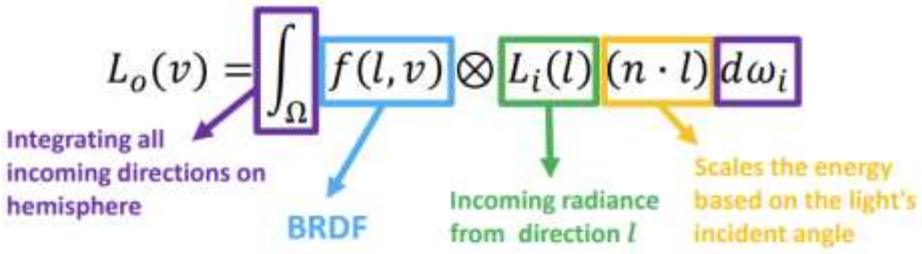


Figure 1 - Traditional Path Tracer


$L_o(v) = \int_\Omega f(l,v) \otimes L_i(l) (n \cdot l) d\omega_i$
Integrating all incoming directions on hemisphere
BRDF
Incoming radiance from direction l
Scales the energy based on the light's incident angle

v: *output direction.*

l: *input direction*

Ω: *full solid angle to integrate*

n: *normal in the surface*

$\otimes$: *component wise multiplication*

Figure 2 - Reflectance equation

The problem of this formula is that we cannot sample the whole hemisphere of the surface, therefore we take advantage of **Monte Carlo Integration**; this means that we will sample each pixel several Monte Carlo integrations and for each time that we hit a surface we will get a random weighted vector based on the BSDF of the surface material to create realistic results.

2. PROBLEM TO SOLVE

The problem with traditional path tracer is that it works very well for infinite area lights like environment maps, but hitting the light is directly dependent on the size of the light and how accessible it is in the scene.

It is true that increasing the number of bounces the chance of hitting the light also increase but the performance suffers as well, but in any case, if the light is very small or is inside a volume like a lantern, even if we increase the bounces, we won't get any better result.

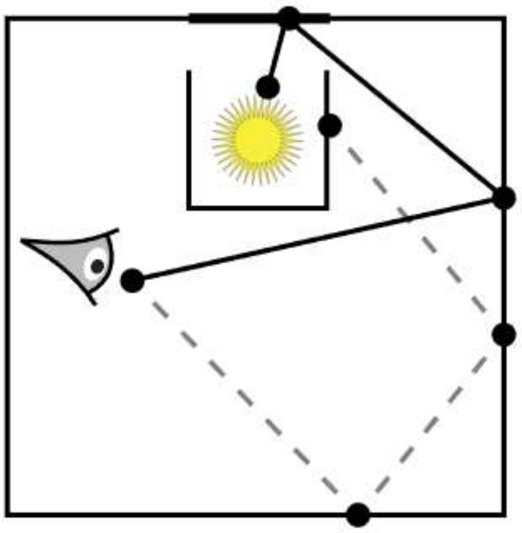


Figure 3 - Path tracing problem

3. SOLUTION: BIDIRECTIONAL PATH TRACING

As the name suggest, the idea behind this technique is to create two paths instead of only one. For each pixel in the screen, we will first create a path the same we did on the traditional path tracer, with the only difference that now we will store each vertex after every bounce on the scene. After we created the first path that will be called the **eye path**, we will create another path sampling the light and throwing random rays to the scene from the light perspective. We will perform the same physically operations to bounce the ray in the scene as the eye path and store the vertex in each bounce creating the **light path**.

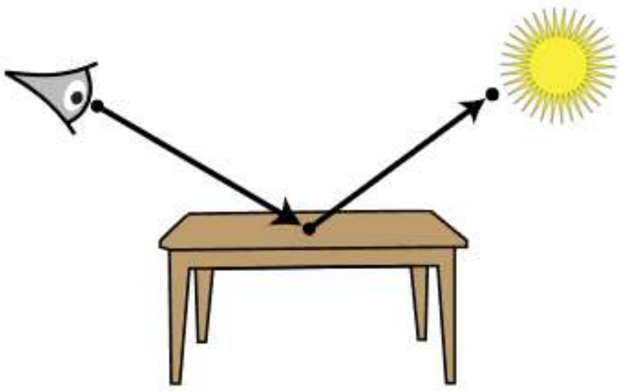


Figure 4 - Eye Path

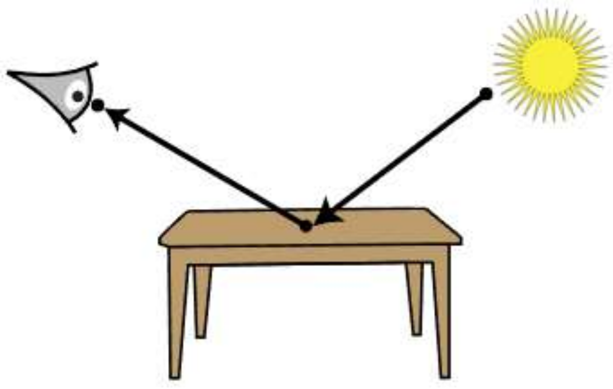


Figure 5 - Light Path

Once both paths are created, we try to connect all the different vertices using shadow rays, which will check if the vertices are visible between them and if that results in positive, then we have a connection, and the appropriate contributions are added to the flux of the pixel in question.

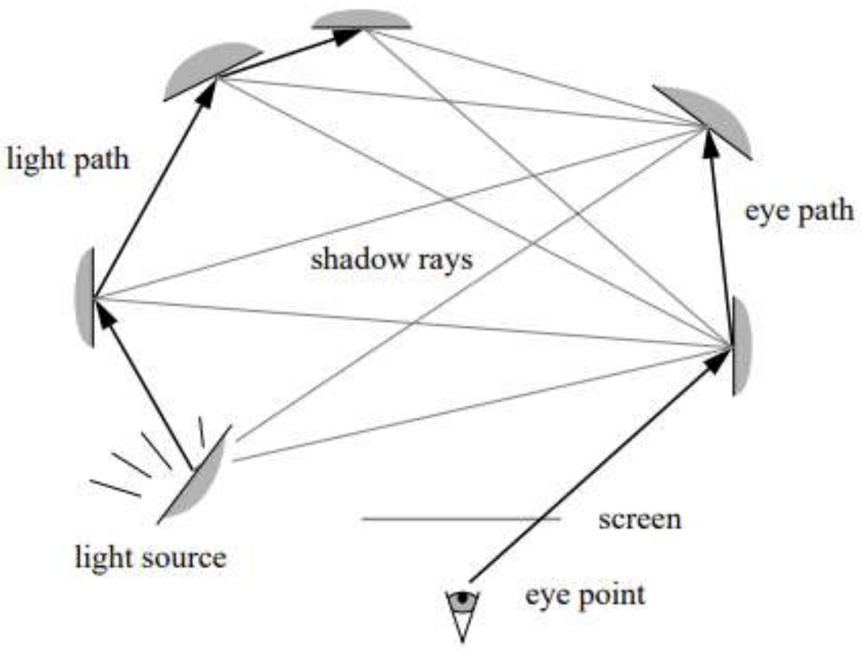


Figure 6 - Connection between paths using shadow rays

4. IMPLEMENTATION

As already explained, we will divide the implementation into three different steps: creating the eye path, creating the light path, and computing the total contribution of flux in the current pixel by connecting both paths.

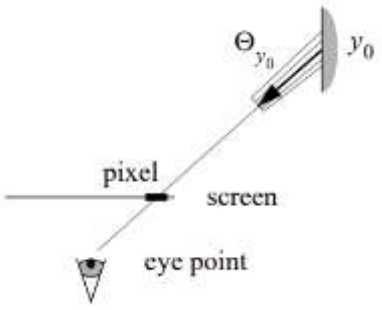


Figure 7 - Eye path Creation

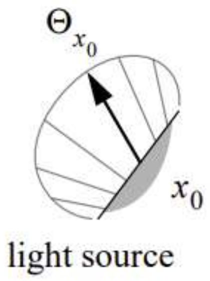


Figure 8 - Light path Creation

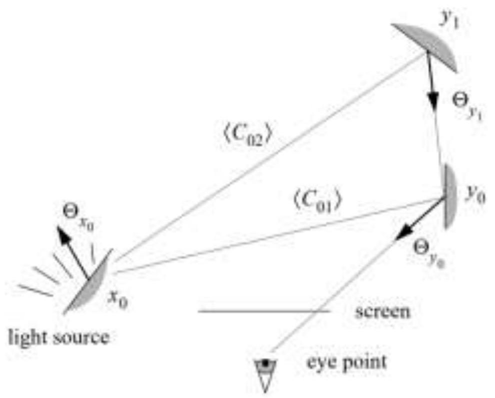
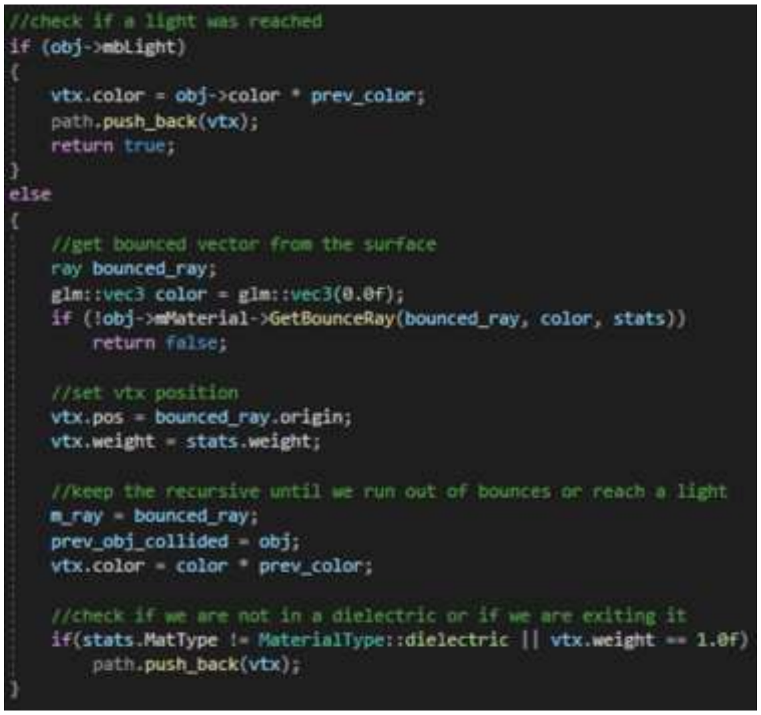


Figure 9 – Connecting both paths

4.1 CREATING THE EYE PATH

We already explained how to create the eye path, but there are some little details that need to be addressed. We will keep track of the previous color of the previous vertex and update it with the following vertices, so when we access it when connecting we already have the color of the full path to get to that vertex.

- We will stop bouncing when reaching the maximum number of bounces or when we hit a light source or an environment map.
- When we hit a diffuse surface, we store the vertex, with the weight on that vertex as 1.0f.
- When we hit a metallic surface, we store the vertex, but the weight on that vertex will be equal to the roughness of the surface.
- When we hit a dielectric surface, we will store the point it we are exiting the object only, because there is no point on storing it when entering and in case of reflection, we are in a metallic material with 0.0f roughness there for it doesn't make sense to store it.



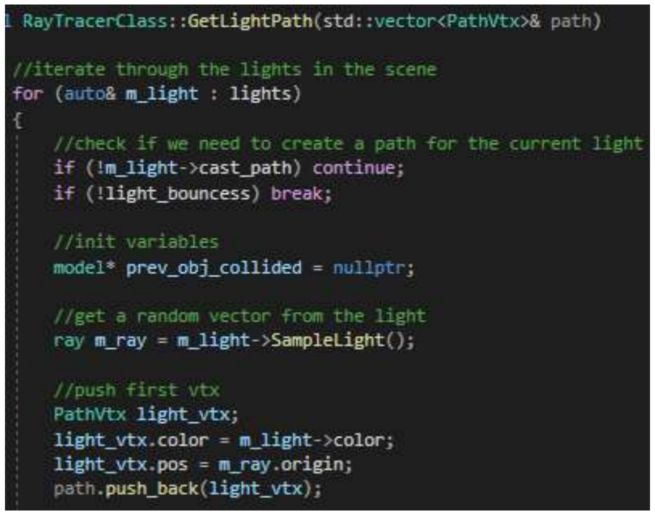
```
//check if a light was reached
if (obj->mbLight)
{
    vtx.color = obj->color * prev_color;
    path.push_back(vtx);
    return true;
}
else
{
    //get bounced vector from the surface
    ray bounced_ray;
    glm::vec3 color = glm::vec3(0.0f);
    if (!obj->mMaterial->GetBounceRay(bounced_ray, color, stats))
        return false;

    //set vtx position
    vtx.pos = bounced_ray.origin;
    vtx.weight = stats.weight;

    //keep the recursive until we run out of bounces or reach a light
    m_ray = bounced_ray;
    prev_obj_collided = obj;
    vtx.color = color * prev_color;

    //check if we are not in a dielectric or if we are exiting it
    if(stats.MatType != MaterialType::dielectric || vtx.weight == 1.0f)
        path.push_back(vtx);
}
```

Figure 10 - Adding a vtx to the eye/light path

4.2 CREATING THE LIGHT PATH

Creating the light path is almost similar to creating the eye path, but with small differences. Now we don't start the ray from a pixel but from a random direction from the light, the idea is to sample the light uniformly. It is important to store the first vertex of the light path that will be in the surface of the light, allowing for direct lighting.

The conditions to add a vertex are the same as the eye path, therefore is the same code as the one in *figure 4*.

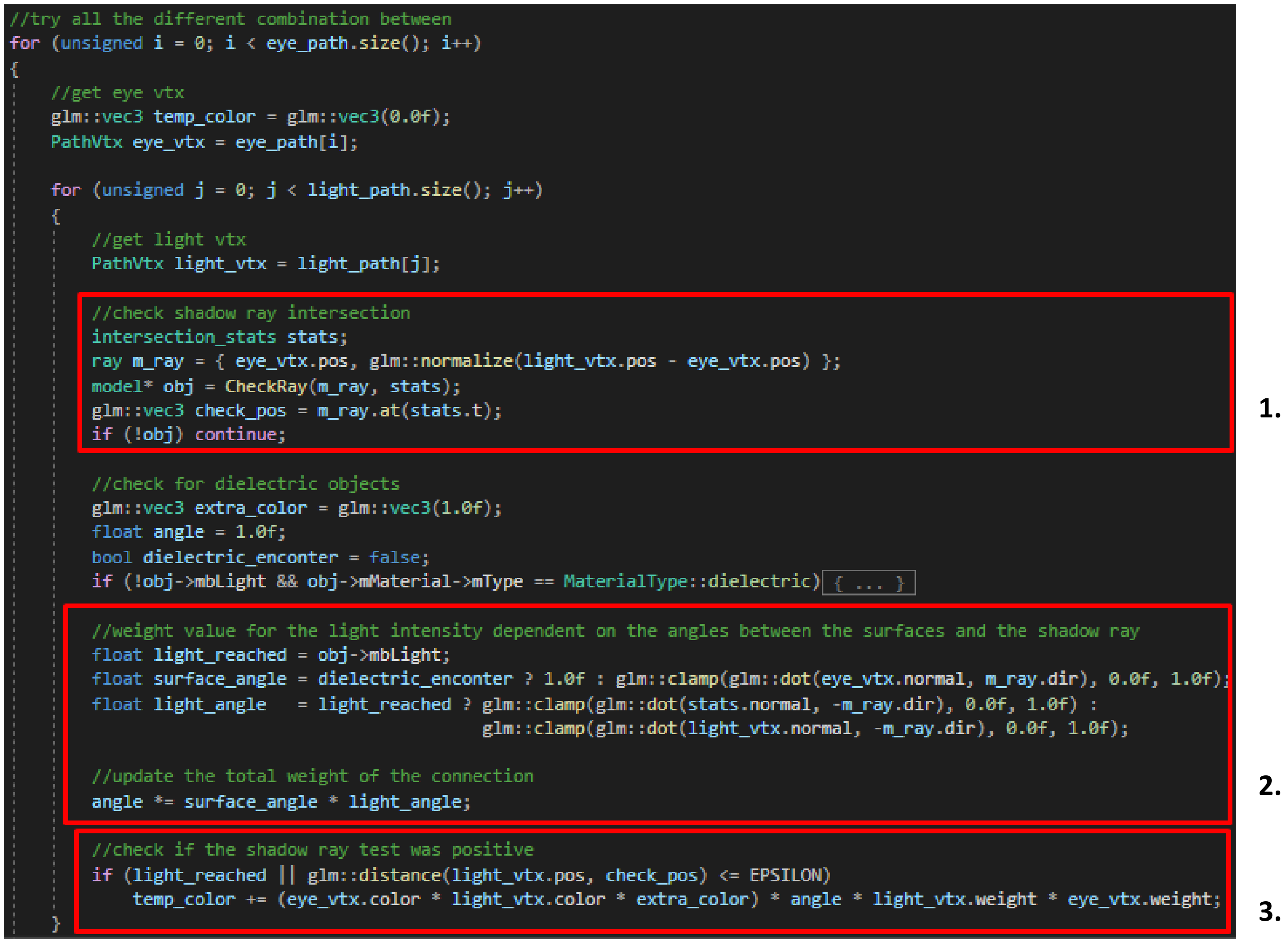


```
l RayTracerClass::GetLightPath(std::vector<PathVtx>& path)

//iterate through the lights in the scene
for (auto& m_light : lights)
{
    //check if we need to create a path for the current light
    if (!m_light->cast_path) continue;
    if (!light_bouncess) break;

    //init variables
    model* prev_obj_collided = nullptr;

    //get a random vector from the light
    ray m_ray = m_light->SampleLight();

    //push first vtx
    PathVtx light_vtx;
    light_vtx.color = m_light->color;
    light_vtx.pos = m_ray.origin;
    path.push_back(light_vtx);
```

Figure 11 - Adding first light vtx for direct lighting

4.3 CONNECTING BOTH PATHS

Once we have all the vertices for both paths stored, we need to try to connect them and get the total flux of the pixel evaluated through the eye path. To do so, we will iterate through both paths trying to connect each vertex using shadow rays, that will test if the object is visible. If the test is positive, we need to average that connection based on the angle between the two surfaces and the shadow ray and then add all the light intensity from each of the path to the total color.



```
//try all the different combination between
for (unsigned i = 0; i < eye_path.size(); i++)
{
    //get eye vtx
    glm::vec3 temp_color = glm::vec3(0.0f);
    PathVtx eye_vtx = eye_path[i];

    for (unsigned j = 0; j < light_path.size(); j++)
    {
        //get light vtx
        PathVtx light_vtx = light_path[j];

        //check shadow ray intersection
        intersection_stats stats;
        ray m_ray = { eye_vtx.pos, glm::normalize(light_vtx.pos - eye_vtx.pos) };
        model* obj = CheckRay(m_ray, stats);
        glm::vec3 check_pos = m_ray.at(stats.t);
        if (!obj) continue;                                                              1.

        //check for dielectric objects
        glm::vec3 extra_color = glm::vec3(1.0f);
        float angle = 1.0f;
        bool dielectric_enconter = false;
        if (!obj->mbLight && obj->mMaterial->mType == MaterialType::dielectric) { ... }

        //weight value for the light intensity dependent on the angles between the surfaces and the shadow ray
        float light_reached = obj->mbLight;
        float surface_angle = dielectric_enconter ? 1.0f : glm::clamp(glm::dot(eye_vtx.normal, m_ray.dir), 0.0f, 1.0f);
        float light_angle   = light_reached ? glm::clamp(glm::dot(stats.normal, -m_ray.dir), 0.0f, 1.0f) :
                                              glm::clamp(glm::dot(light_vtx.normal, -m_ray.dir), 0.0f, 1.0f);

        //update the total weight of the connection
        angle *= surface_angle * light_angle;                                            2.

        //check if the shadow ray test was positive
        if (light_reached || glm::distance(light_vtx.pos, check_pos) <= EPSILON)
            temp_color += (eye_vtx.color * light_vtx.color * extra_color) * angle * light_vtx.weight * eye_vtx.weight;   3.
    }
```

Figure 12 - Connecting both paths and adding total contribution

1. Shadow ray test.
2. Weighting the connection.
3. Adding the total contribution of both paths connected with the respective weights.

5. RESULTS

Here we will show the results comparing the traditional path tracing and the bidirectional path tracing. For the traditional path tracing we have a maximum number of 24 bounces, while the bidirectional path tracing has a maximum number of 4 bounces for the light path and another 4 for the eye path.

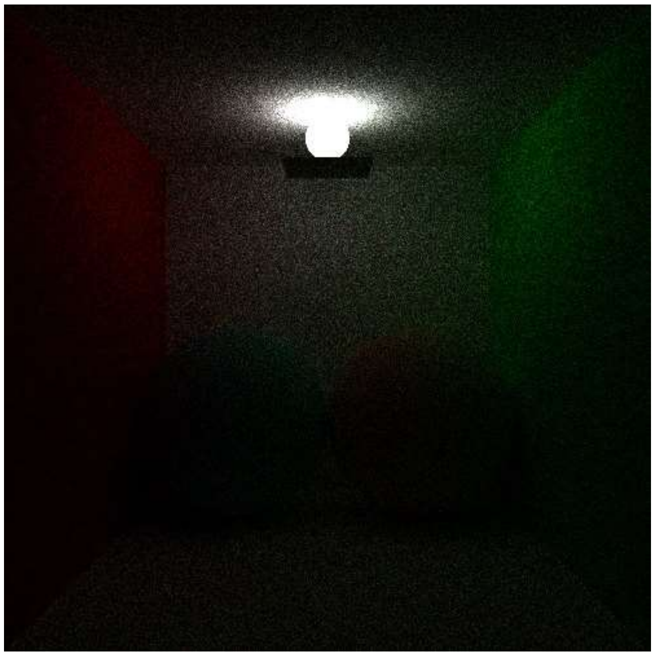


Figure 13 - Traditional 250 samples

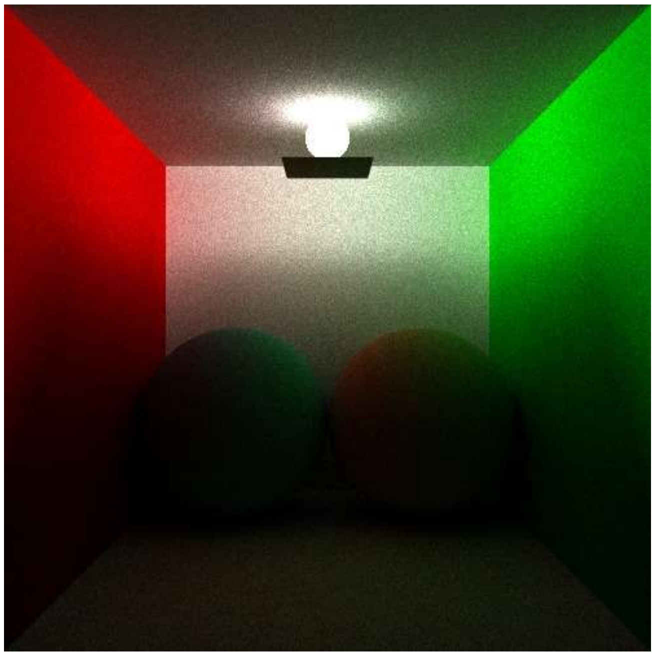


Figure 14 - Bidirectional 250 samples

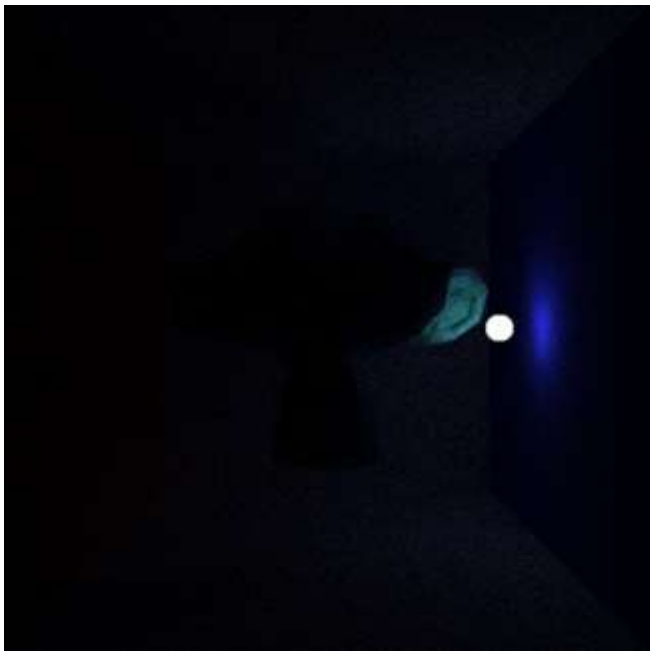


Figure 15 - Traditional 500 samples

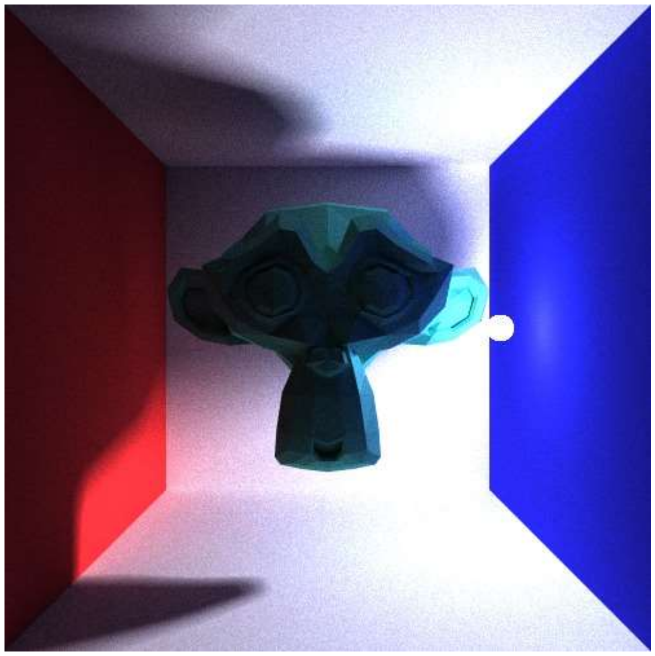
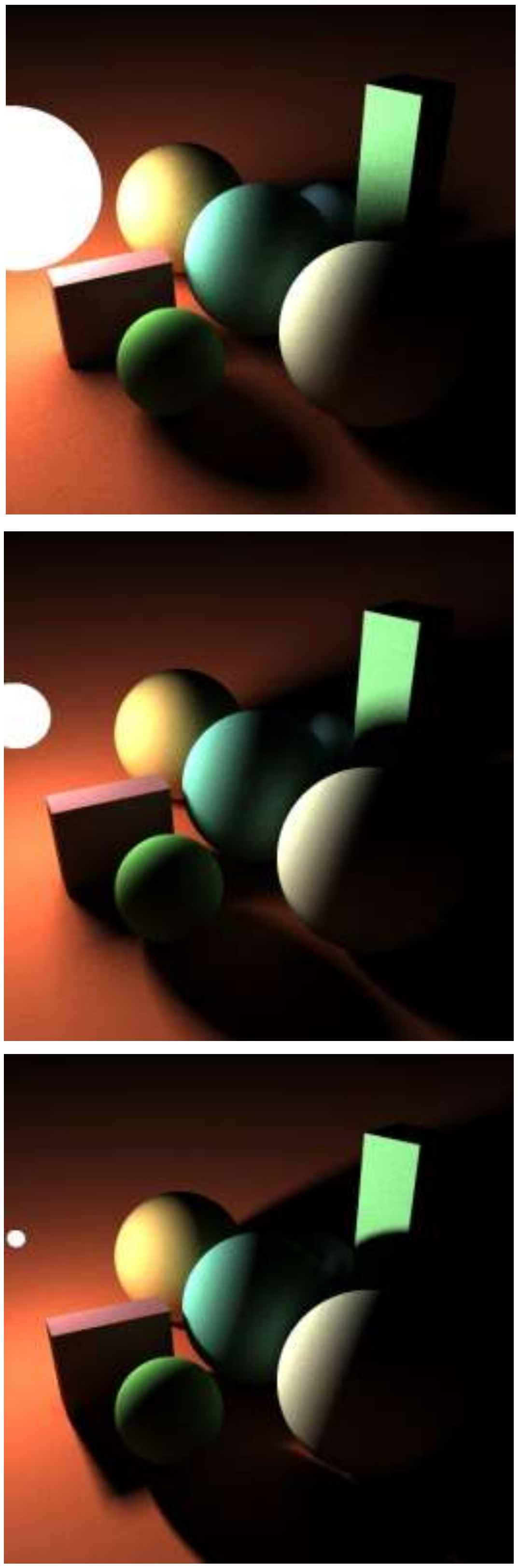


Figure 16 - Bidirectional 500 samples



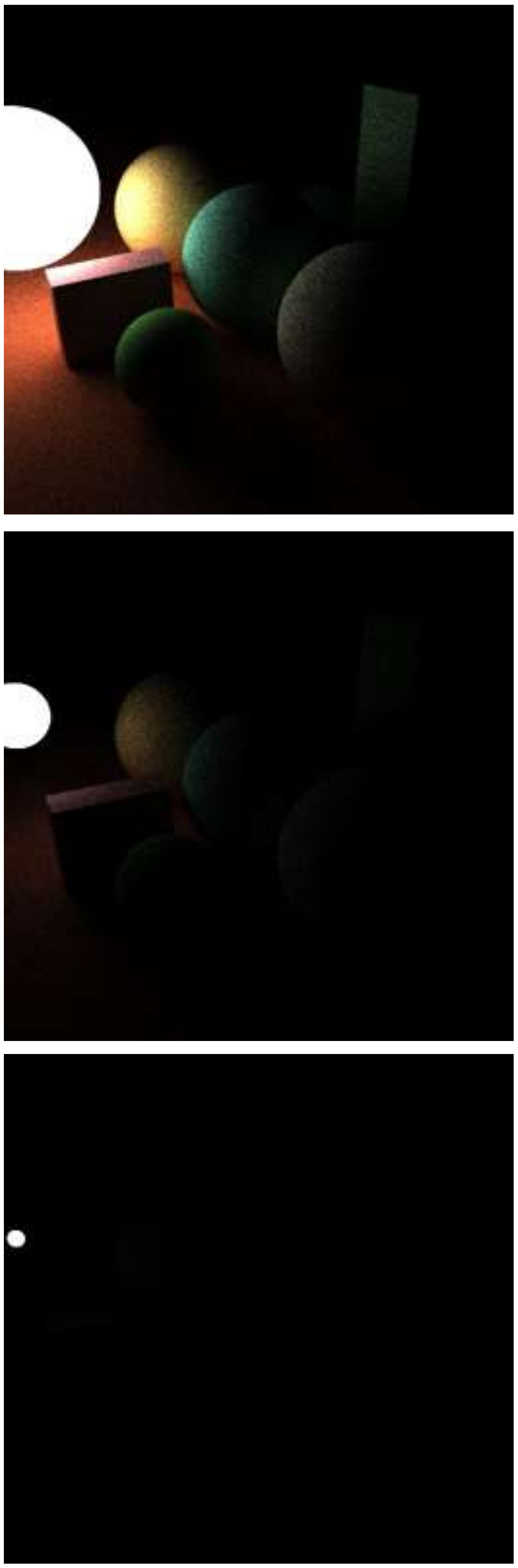


Figure 17 - Bidirectional 200 samples

Figure 18 - Traditional 200 samples

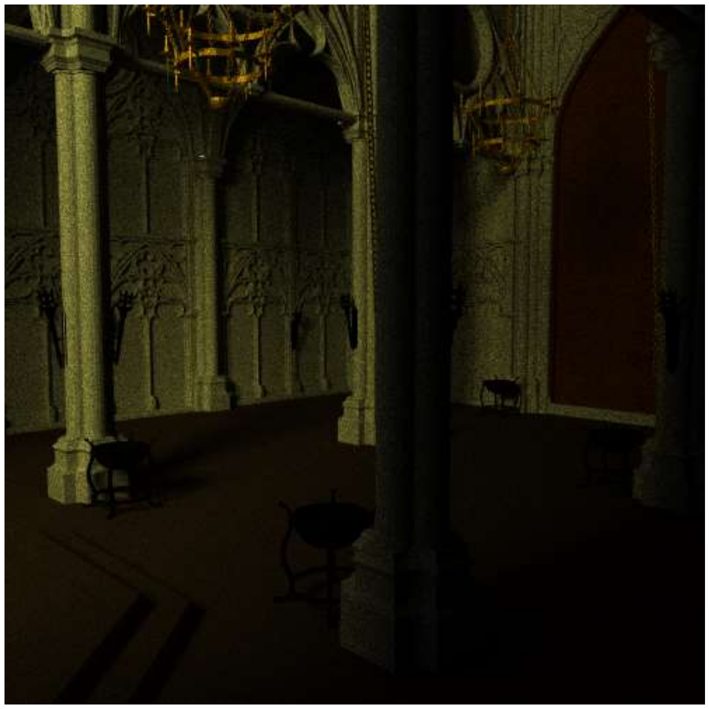


Figure 19 - Complex Scene Bidirectional 100 samples (10 minutes)

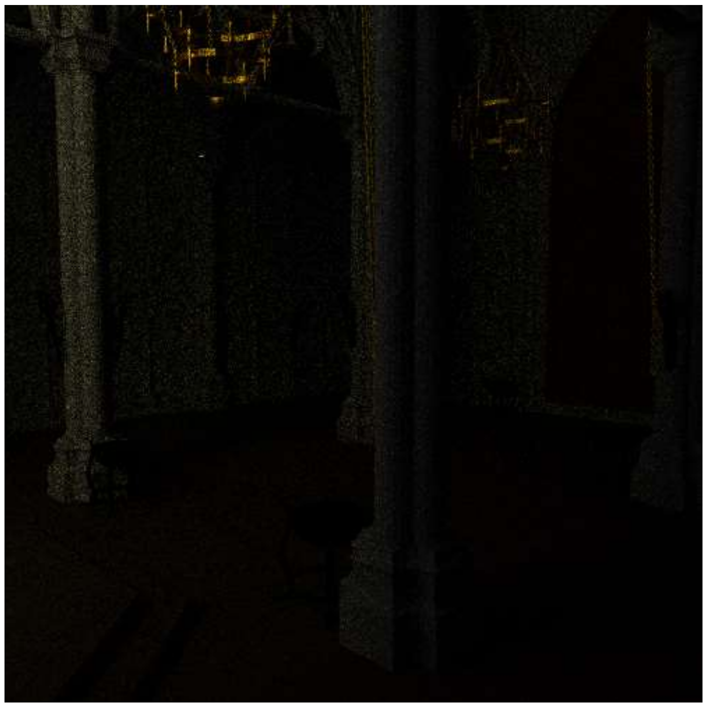


Figure 20 - Complex Scene Traditional 100 samples (20 minutes)

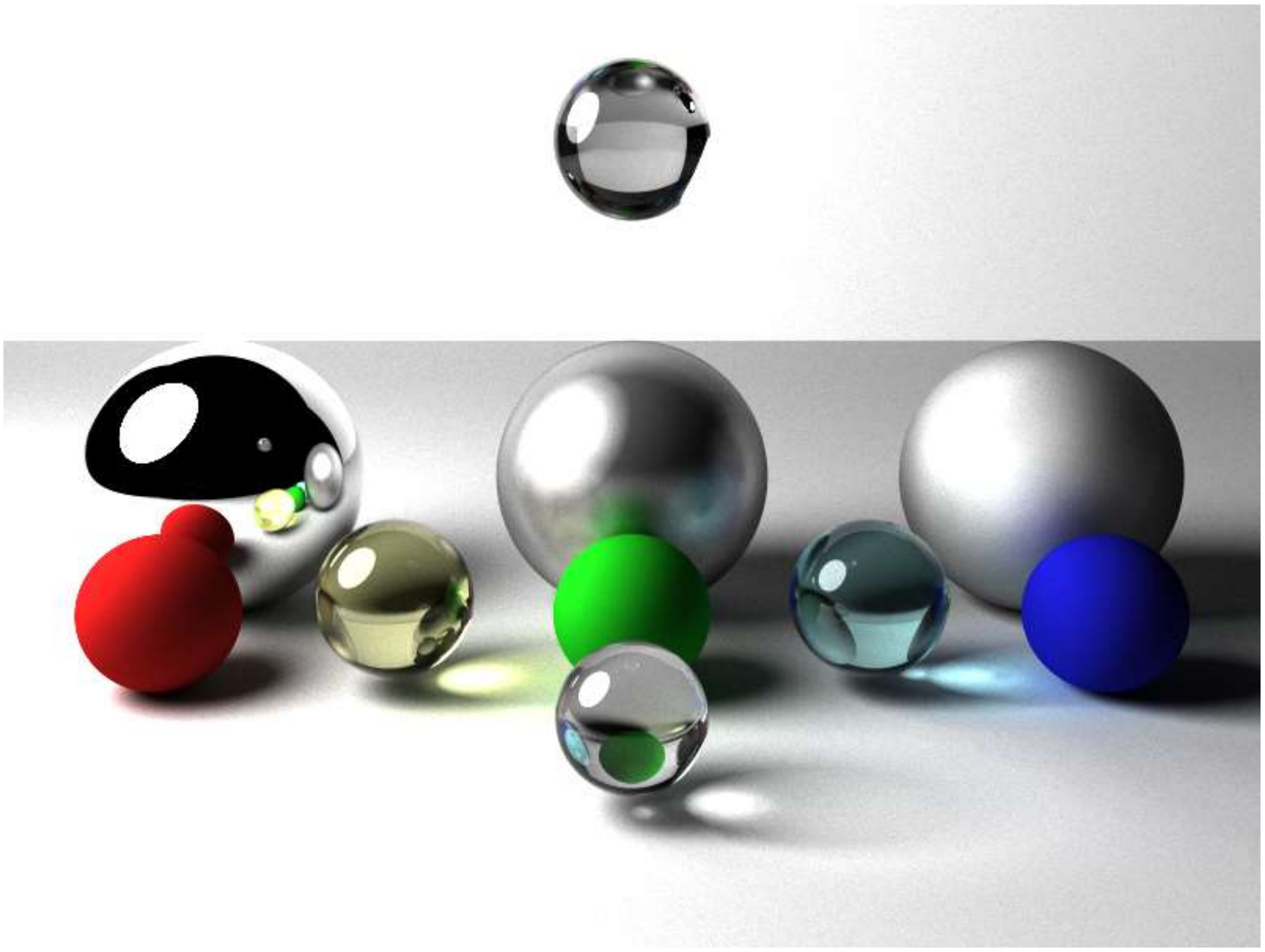


Figure 21 - Bidirectional caustics 2000 samples

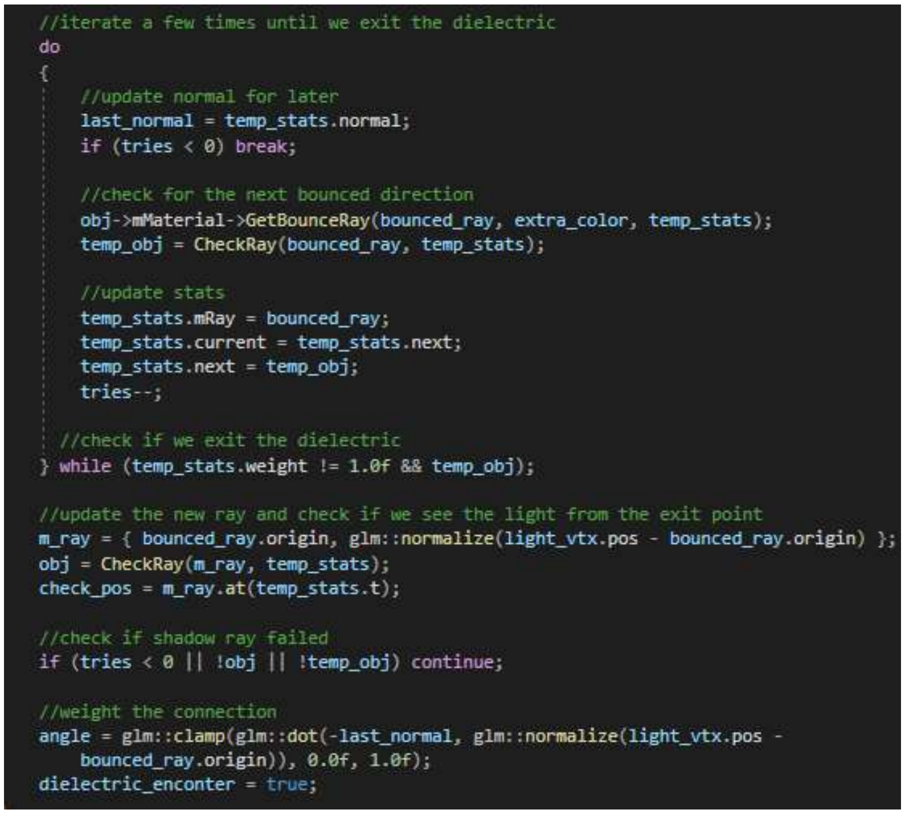
6. HOW TO USE DEMO

- There are default values for the program, but for a custom scene, screen shot and dimensions it needs to follow the following convention in the command line: "scene_name.txt" "screenshotname.png" "width" "height".
- The user can change different parameters of the demo by changing the values of the variables in "config.txt".
- The user only needs to run the program, the user can take a screen shot pressing F1, and reload the image be pressing R.

7. IMPROVEMENTS / PROBLEMS

There are other techniques that also try to solve the same problem, such as **Photon Mapping** and **Metropolis Light Transport**. Photon Mapping is more focused on getting better caustics effects than Bidirectional Path tracing.

A problem that I encountered was that since there wasn't any information on how to handle shadow rays that collide with dielectric objects. If we treat them as normal objects and get a negative check for the shadow rays all the caustics of the scene will lose a lot of intensity. In order to solve this, my solution was that whenever a shadow ray hits a dielectric object, I will traverse the dielectric until I exit it and then check the with the exiting point of the shadow ray. This solution gave very nice results, but an artifact appears that I haven't been able to fix yet.



```
//iterate a few times until we exit the dielectric
do
{
    //update normal for later
    last_normal = temp_stats.normal;
    if (tries < 0) break;

    //check for the next bounced direction
    obj->mMaterial->GetBounceRay(bounced_ray, extra_color, temp_stats);
    temp_obj = CheckRay(bounced_ray, temp_stats);

    //update stats
    temp_stats.mRay = bounced_ray;
    temp_stats.current = temp_stats.next;
    temp_stats.next = temp_obj;
    tries--;

  //check if we exit the dielectric
} while (temp_stats.weight != 1.0f && temp_obj);

//update the new ray and check if we see the light from the exit point
m_ray = { bounced_ray.origin, glm::normalize(light_vtx.pos - bounced_ray.origin) };
obj = CheckRay(m_ray, temp_stats);
check_pos = m_ray.at(temp_stats.t);

//check if shadow ray failed
if (tries < 0 || !obj || !temp_obj) continue;

//weight the connection
angle = glm::clamp(glm::dot(-last_normal, glm::normalize(light_vtx.pos -
    bounced_ray.origin)), 0.0f, 1.0f);
dielectric_enconter = true;
```

Figure 21 - Traversing dielectrics with for shadow rays

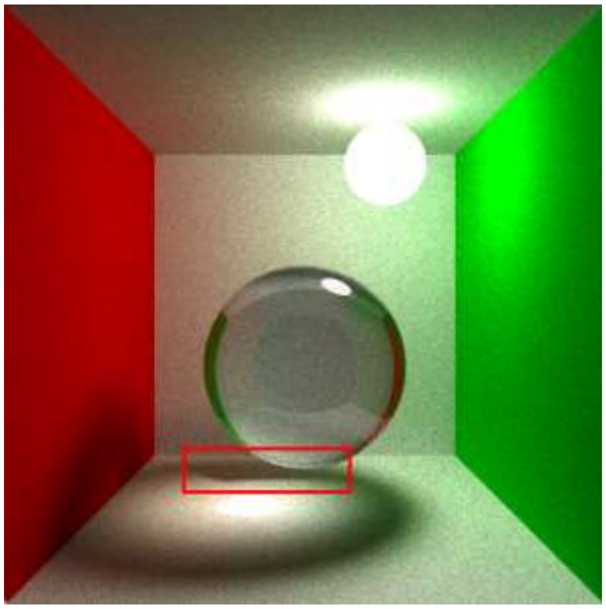


Figure 22 - Solution with artifact

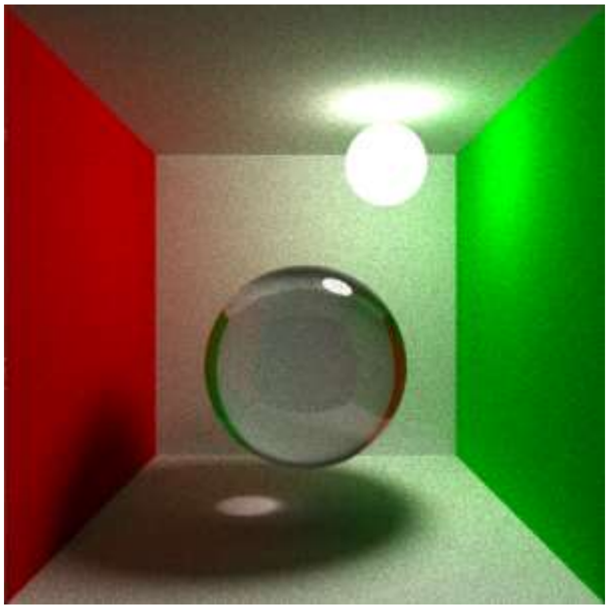


Figure 23 – Not solution with normal shadow rays

Another problem that I have in the implementation is that sometimes the demo restarts or gets bugged, it only happens when the application is running in the background, therefore I suspect that it has something to do with multithreading, but I can't confirm it.

8. CONCLUSIONS

- It works very good for small lights and lights that are difficult to reach.
- It isn't as good as photon mapping for caustics.
- It is difficult to handle shadow rays with dielectric objects in the middle.
- It doesn't work for infinite area lights or environment maps.
- The idea of the algorithm is easy to understand, but when you reach all the probabilities that you need to take into account and start merging different kinds of materials, the complexity grows a lot.

9. BIBLIOGRAPHY

Lafortune, E.P. & Willems, Y.D. *Bi-directional Path Tracing*. Catholic Leuven University.

Pharr, M., Jakob, W., & Humphreys, G. (2018). Chapter 16.3 *Bidirectional Path Tracing* [Libro electrónico]. En Physically Based Rendering: From Theory to Implementation (3.a ed.). Morgan Kaufmann.

Veach, E. Chapter 10: *Bidirectional Path Tracing*. In: Veach, E. (1997). Robust Monte Carlo Methods for Light Transport Simulation. Standford University