Raytracer Project

CR11 course evaluation

Fabrice Lebeau

December 15, 2017
ENS de Lyon

Contents

Introduction

In this report, I present the functionalities I implemented for the project of the CR11 course. I chose to work on a *raytracer*: a physically-based random generator simulating light rays for rendering a 3D scene. I chose to implement it in `C++`, both for performances and because it is the language I feel the most comfortable with.

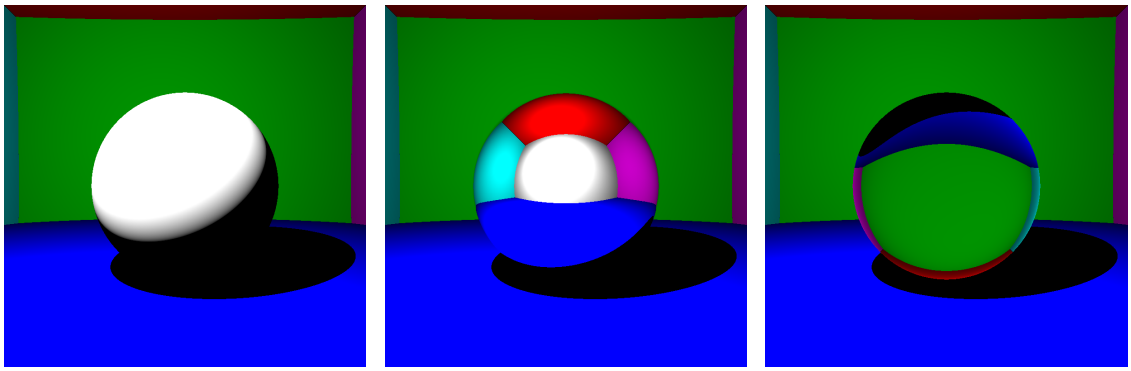
I give a few examples of generated images in this report, they have all been generated with my program on the same computer and with the same resolution (800x800), so that the computation times can be compared.

1 Basics of a raytracer

In my raytracer program, a scene is a set of *spheres* (and only spheres) given by their center, radius and a *material*.

At first, the material was just a color given by RGB coefficients (see Figure 1a). Then I implemented reflexion, so that a material can be a mirror or partially a mirror, with a reflexion coefficient and reflexion color (see Figure 1b). Then I added refraction, so a material can be partially transparent with a coefficient, and a refraction color (see Figure 1c). At this point, the

expressivity of materials is quite good, since all combinations of mirror/transparent/concrete objects are possible (even the less realistic like a third blue-diffusive object, a third red-refractive and third green-reflexive – see Figure 2).



(a) Concrete sphere (b) Mirror sphere (c) Glass sphere (index 1.5)

Figure 1: Basic raytracer features – computation time $\leq$ 1 sec

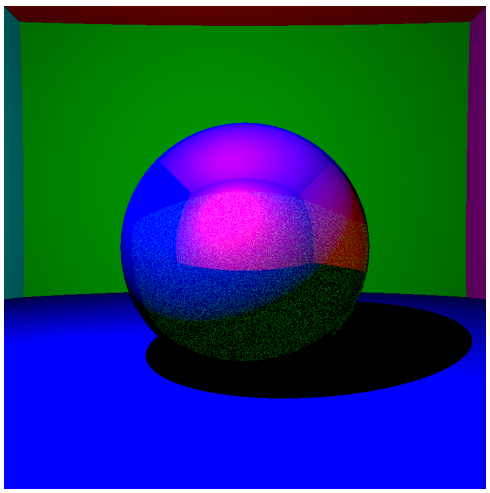
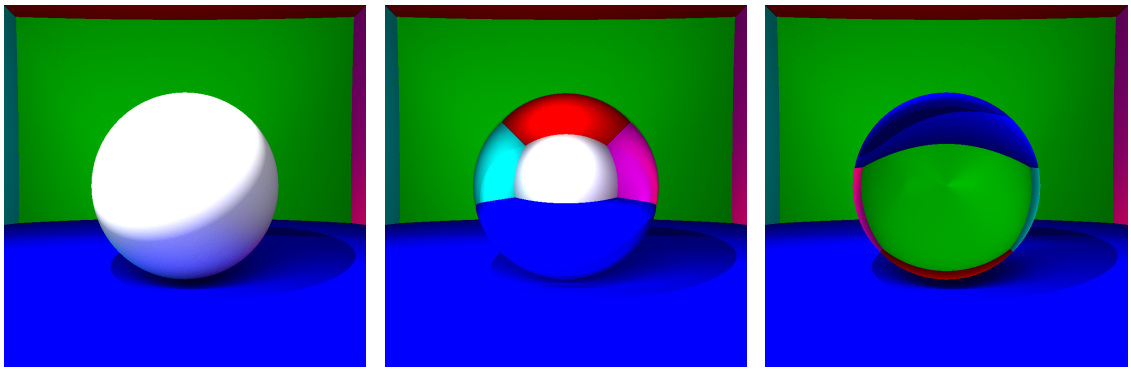


Figure 2: Weird material – computation time $\leq$ 1 sec

I also implemented *diffuse lightning*: it is a partial reflexion of the color of diffuse object (with a diffuse coefficient and color). It generates smoother shadows and the appearance of close colors on objects (see Figure 3). However, the diffusion step introduce randomness in my program: we pick a direction randomly (uniformly) for the bouncing ray. This requires to launch a lot of rays for each pixel, which increases the computation time dramastically. In all the following examples I always used 1000 rays per pixel, and 12 bounces, which gives nice results.

2 Antialiasing and Fresnel coefficients

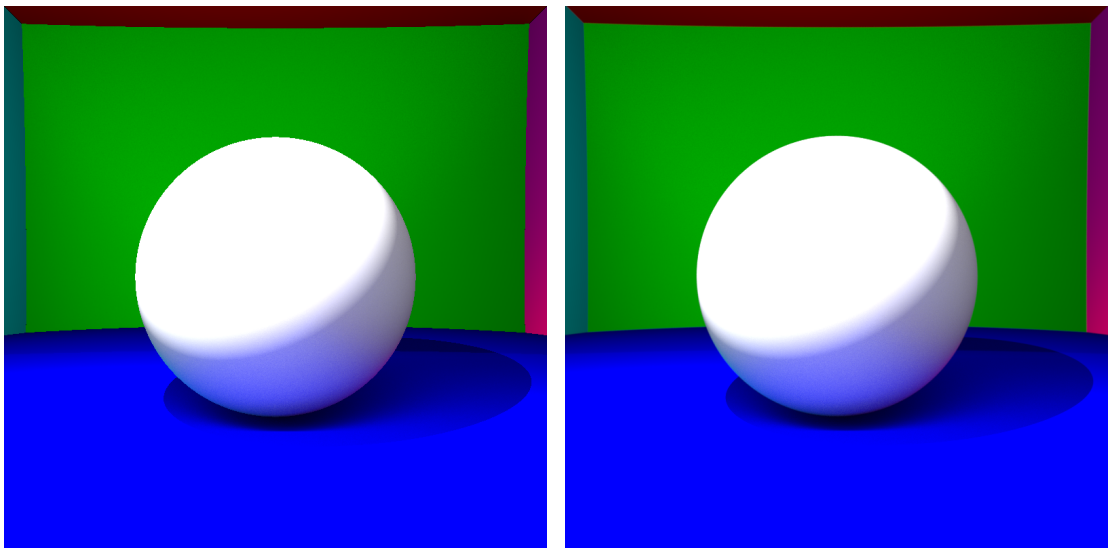
One the images generated so far, one can see that the edges are not very smooth, they have a "stair" shape (it is particularly clear on the edges of the green wall in Figure 4a). To obtain



(a) Concrete sphere (b) Mirror sphere (c) Glass sphere (index 1.5)

Figure 3: Basic scenes with diffusion – computation time $\approx$ 30 min

smoother edges for the scenes, I implemented an antialiasing feature, which introduce randomness (a gaussian perturbation) in the direction we launch each ray for each pixel (see result in Figure 4b). We can see that the edges are a lot smoother. Remark that the computation time increases only by 1 minute on a total of 30 minutes, so the increase for antialiasing is very small. In all the following images, antialiasing is enabled.

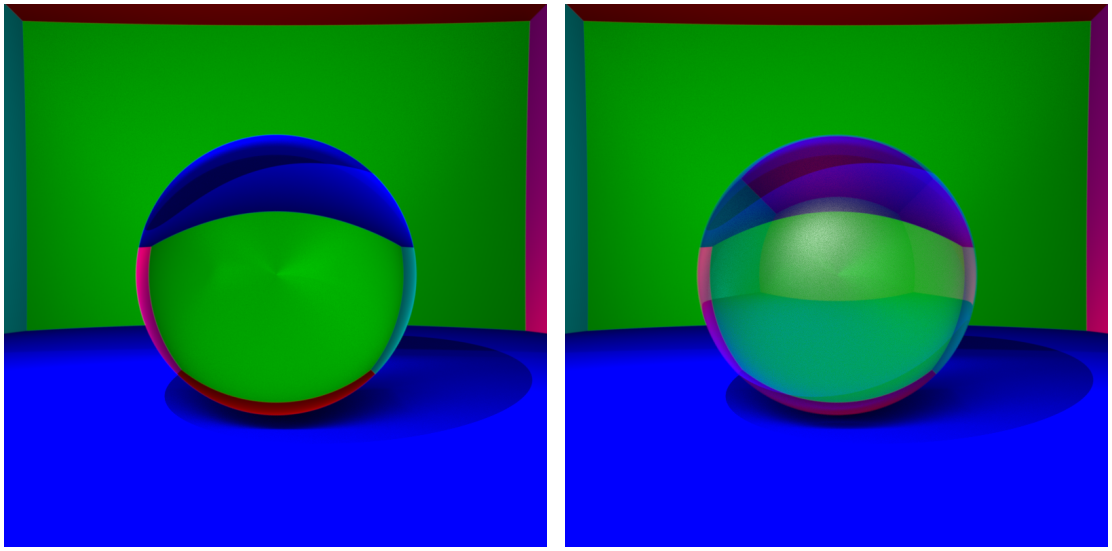


(a) Without antialiasing – $\approx$ 30 min (b) With Antialiasing – $\approx$ 31 min

Figure 4: Comparison with and without antialiasing

The next improvement that I implemented was the use of Fresnel coefficients for completely transparent spheres (no user-defined specularity or diffusion). If you look at Figure 5a, you will see that the glass sphere is completely transparent. This seems unrealistic to a human eye, as the light which is above the camera should reflect partially in the glass sphere. With the Fresnel coefficients, depending on the refractive indices of the in and out materials, and depending on the angle of the incoming ray, sometimes the ray will be reflected and sometimes it will be refracted, which gives a more realistic image (see Figure 5b) as the light and the other side of the room are partially reflected by the glass sphere. Remark that the computation

time is nearly the same with or without this improvement, which means that the computation of Fresnel coefficients is not very costly. In all the following images, Fresnel coefficients are enabled.



(a) Without Fresnel coefficients – ≈ 31 min (b) With Fresnel coefficients – ≈ 32 min

Figure 5: Fresnel coefficients

3 Inclusion of transparent spheres

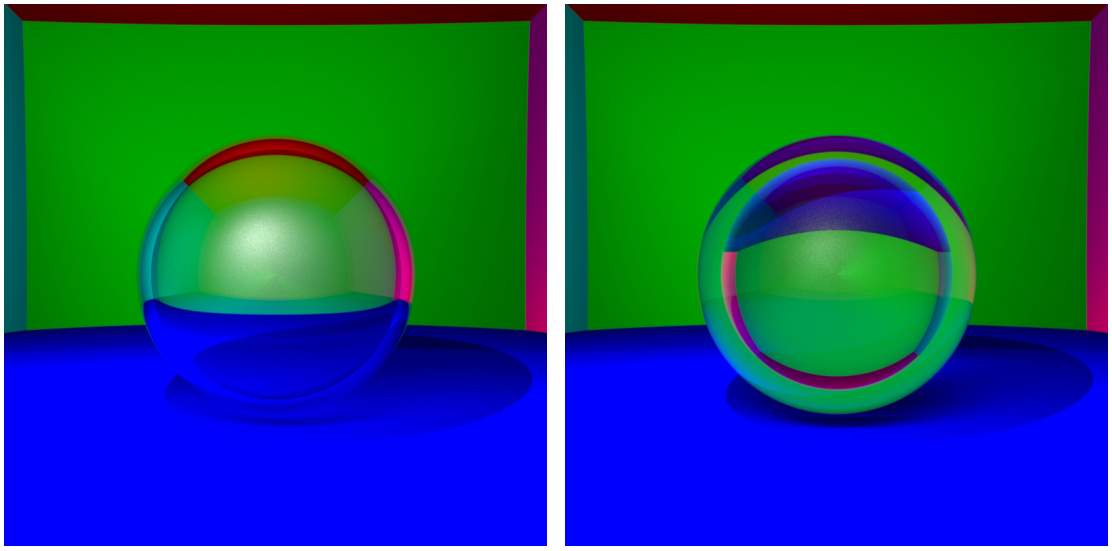
While implementing the refraction functionality, I first considered that the inside refractive index of a sphere is given in the material information, and that the outside refractive index would always be 1. However, I realized that it was not always the case: a sphere of air (of index 1) in a sphere of glass (of index 1.5) has outside refractive index of 1.5, and not 1. In fact, I found another issue: what happens when two transparent spheres have a non-empty intersection but without one being included into the other? In that case, the refractive index of the intersection is not defined: it could be the sum, or the index of one of the two spheres...

In order to tackle this issue, I decided to restrict the input scenes, by adding the following requirement: *two transparent spheres are either disjoint or one is included into the other*. Then, I needed to implement a data structure that would give for every transparent sphere the refractive index of the outside of the sphere. My method only requires a precomputation step before the simulation. It works as follows:

- spheres of the scene are indexed $0, 1, 2..., n$, sorted by *decreasing radius*;
- a vector `sphere_inclusion` is filled by a precomputation step so that the i-th value in the vector is the index of the smallest sphere containing the i-th sphere (it is equal to i if the i-th sphere is included in no other sphere);
- using vector `sphere_inclusion`, one can easily determine the outside refractive index of a transparent sphere: the outside index of sphere i is 1 if the i-th value of the vector is i (we assume default is air), otherwise it is the refractive index of the sphere whose index is the i-th value in `sphere_inclusion`.

Let me now explain the precomputation step. Since spheres are sorted by decreasing radius, to compute the i-th value of vector `sphere_inclusion`, it suffises to take the largest $j < i$ such that the i-th sphere is included in the j-th sphere if there exists such j (otherwise the vector was already initialized with value i). Checking the inclusion of spheres is a simple condition on the radiuses and the distance between centers.

With this improvement, it is possible to consider interesting scene, like a thin "glass bubble" (air sphere in a glass sphere – see Figure 6a), or a sphere of glass into a sphere of lighter glass (Figure 6b).



(a) Glass bubble – ≈ 33 min

(b) Glass in lighter glass (index 1.2) – ≈ 32 min

Figure 6: Inclusion of transparent spheres

4 Other types of material and scene parser

At some point I wanted to improve the modularity of my code, in order to allow other types of concrete materials. More precisely, I changed my class `Sphere` so that it contains a virtual method `color` which takes as input a point P (assumed on the sphere) and returns the color of point P. Then, one can define his or her own material by making a class that inherits from class `Sphere` and implement custom `color` method. I implemented a small example to illustrate this: a multicolor sphere that divides the sphere into eight "quarters" colored differently (Figure 7).

At this point in the development of my project, I was still hard coding scenes in the main function of the program and compile everytime I wanted to change the scene, which is not a very clean way of doing things. That is why I implemented a simple scene parser (the specification can be found in the `README` file), so that I can store the scenes I use (they are in the `scenes` folder) and generate images without compiling my program everytime. I also added a Bash script called `generate_images.sh` that generate all the images used in this report (it takes a few hours...).

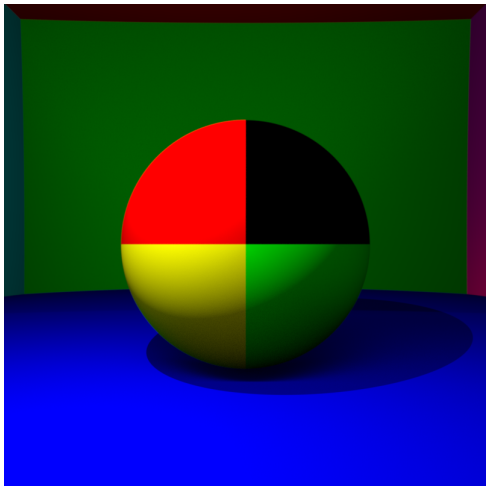


Figure 7: Multicolor sphere – ≈ 32 min

5 Random scene generator

Before implementing other functionalities (that I did not have time to implement in the end...), I wanted to see how my raytracer was doing on bigger scenes that the ones I considered until now. I figured that the easiest way to produce a big scene would be to implement a random scene generator. By making a hundred random scenes with fast rendering with my raytracer, I was able to select a few interesting scenes that shows the functionalities of my project.

The random generator works as follows:

- the user chooses: the size of the domain of the scene (which will be encased in walls), the number of spheres, the probability of getting a concrete, mirror or transparent sphere and the minimum and max radius of a sphere;
- then the generator incrementally tries to create random spheres by uniformly choosing the center in the domain, a radius which is between the specified minimum radius and a maximum radius, and a material according to te specified probabilities (the color of concrete materials are choosen randomly). The maximum possible radius of the sphere is computed so that it is less tan the specified maximum radius, and such that the sphere is either contained or disjoint from any other created spheres.

The examples that are given below (Figure 8 and Figure 9) are generated with a cubic domain of side 80, minimum radius 5 and maximum radius 10 and probabilities 0.5 of being transparent, 0.35 of being a mirror, 0.15 of being concrete. If the scene themselves are very unrealistic (maybe even psychedelic?), I think the rendering is interesting and quite realistic, which illustrates that when it comes to rendering spheres my raytracer is quite good. Indeed, I particularly notice the fact that mirror spheres print a “mark” on the walls (for instance bottom left of Figure 8d), and the very complex refracted shapes that appears (for instance the big foreground sphere in Figure 8a). However, the computation time for these images is *very large*, between 45 minutes and more than an hour...

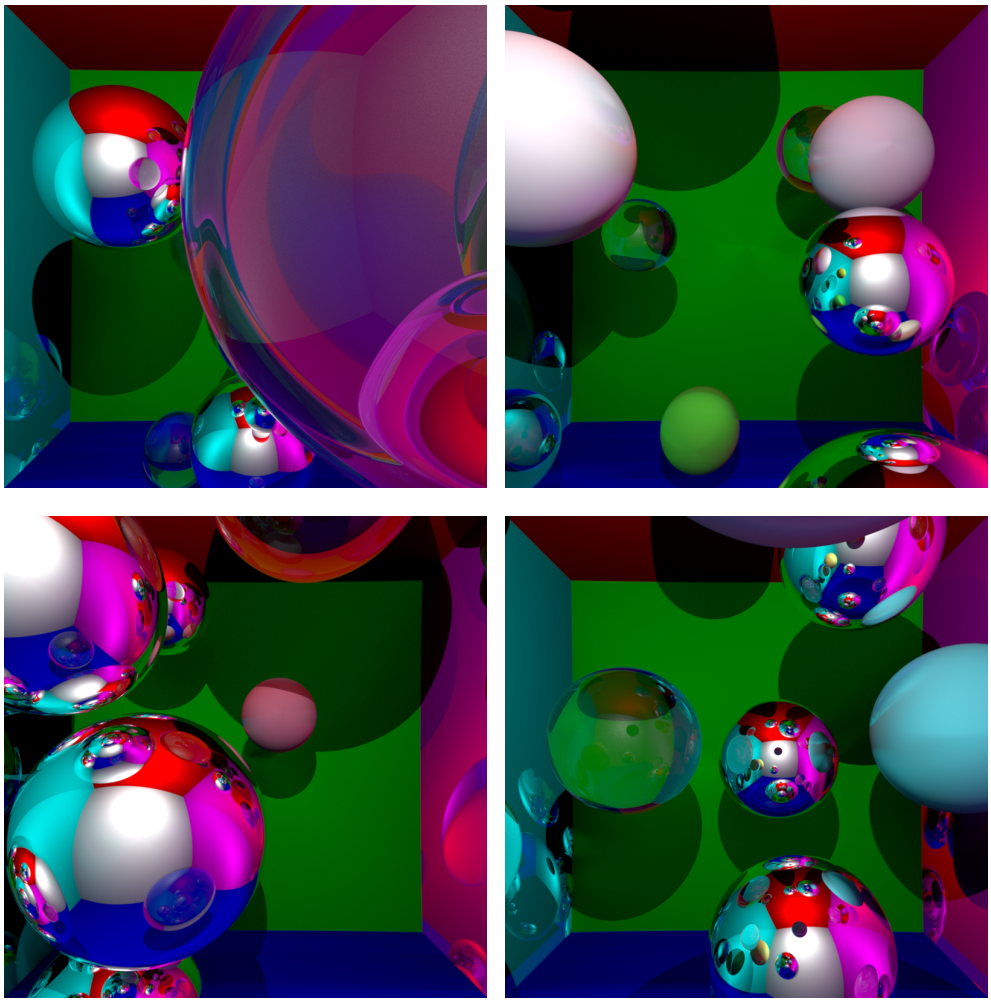


Figure 8: Random scenes with 15 spheres – 45-50 min

Conclusion

I have implemented a raytracer with several functionalities that gives quite realistic results (diffusion, antialiasing, fresnel coefficients) and a random scene generator to test my program on big examples. However there is still a lot of room for improvements:

- I have not been extremely careful about performances: it must be possible to simplify operations (norm computations for example) in order to speed-up my program;
- I have only considered scenes with *sphere*, which is very limited. In order to import 3D models one would have to handle triangles. My raytracer could not be used as it is to render useful scenes that we are interested in in practice.

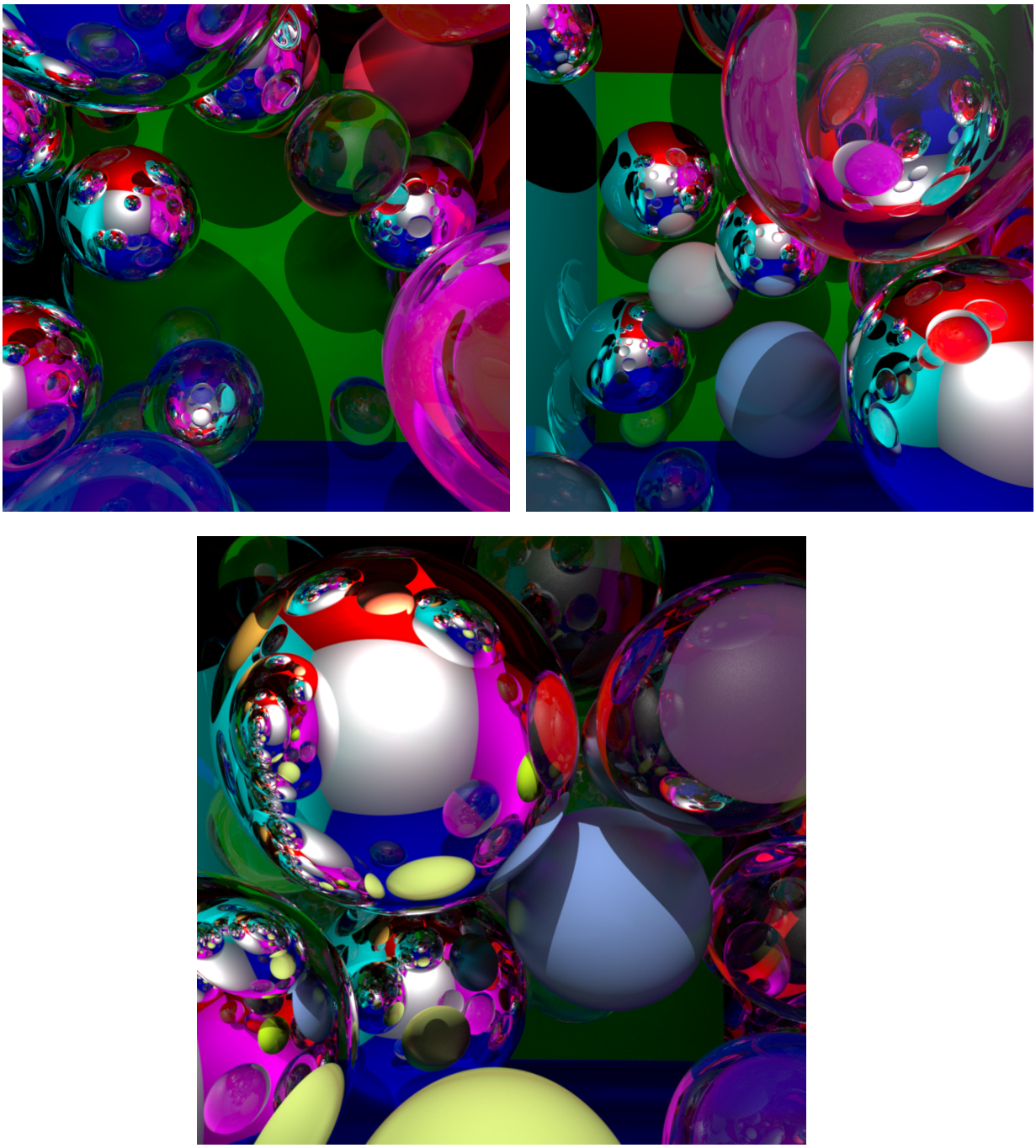


Figure 9: Random scenes with 40 spheres – 70-80 min