Ray-tracing based renderer from scratch in Python

Jörn Lasse Vaupel

February 20, 2023

Contents

Abstract

1 Introduction

This report summarizes the process of implementing a ray-tracing based renderer from scratch following the introduction from Peter Shirley's Ray Tracing in One Weekend [1] and its continuation Ray Tracing: The Next Week [2]. While Shirley used C++ as a high performance and efficiency programming language, the following implementation is written in Python. Even if the rendering times may be longer, Python's readability and easiness helps focusing on the basic concepts of ray-tracing based rendering. In addition, efficiency might be higher if the advantages of Python libraries like *NumPy* would be used more strictly, but the focus of the implementation is on understanding, not on efficiency. All in all, some functions are just easier in Python than in C++.
In the working process many different test images were rendered. In order to meet all the requirements for the final result and generate a uniformly growing scene, the implementation was completed before rendering the final images taking into account a backwards compatibility. Therefore, all final images can be generated by using the same structure: a scene containing one or several cameras and some spheres. Only the single-colored image is generated differently due to the gamma correction. This shows some design decisions in the implementation like a horizontal Field of View (FOV) or gamma correction, can lead to slightly different results than those, that were possible in previous stages of implementation. This is no big deal but should be mentioned.

2 First rendering loop

Just with the beginning the first example of understanding against efficiency occurs. In order to speed up rendering and to limit the amount of file accesses, the rendered image is saved in an image array maintained by an *Image* class before saved in a .ppm file. The rendering loop requires a double for loop. To avoid generating the file from scratch within another double for loop, it's possible to use Python libraries like *Pillow* to directly write a .ppm file. But to enable a deeper understanding of the structure of the file format while utilizing the advantages of Python libraries, a custom approach based on *NumPy* is used (the double for loop approach is also shown but not used).
The code from *scene1.py* in listing 1 shows the basic use of the *Image* class and the render loop which generates the image in fig. 1. The render loop will be later handled by the *Camera* class.

Listing 1: Example for a basic render loop

```
1 image = Image(1920, 1080)
2 for y in range(image.height)[::-1]:
3         for x in range(image.width):
4                 image.image_list[y, x] = Vector(128, 64, 255).to_int_array()
5
6 image.save_image("../images/image1.ppm")
```

Figure 1: Singe-colored image generated by first render loop

3 Camera

The implementation of the camera in the *Camera* class largely follows the tutorial in [1]. In general, it's necessary to define a large set of algebraic methods within the *Vector* and *Ray* class. Referring the *Vector* class, it's important to mention that the default operators (e.g. + and -) are overloaded and especially the meaning of * operator depends on the second operand. If it's a float the operation defines a scalar multiplication, if it's a vector the result is the dot product of both vectors.
A small addition to the concept in [1] used in this implementation is the idea that every object within a 3D scene has a position (and a rotation) and can therefore be described as an object of a common class: the *Transform* class. This does not grant any advantages now but makes the implementation of new 3D objects like spheres (or maybe a collection of planes in future) easier.
The resulting image is shown in fig. 2. Because of the lag of backwards compatibility described in section 1 or - to be more precisely - due to personal favor, the FOV is slightly different in all rendered images. The default horizontal FOV is set to 130° (representing human horizontal FOV), while the FOV ϕ in [1] results from focal length $fl = 1$, viewport height $vp_h = 2$ and aspect ratio $ar = 16/9$ as shown in eq. (1).

$$\phi = 2 \cdot arctan\left(\frac{vp_h \cdot ar}{2 \cdot fl}\right) \approx 121,28° \tag{1}$$

4 Spheres

Just like the implementation of the camera the implementation of spheres within the *Sphere* class was straight forward understanding the concepts from [1] and transfer them to Python. Key feature is the *hit(ray, t_min, t_max)* function

Figure 2: Gradient image generated by camera (empty 3D scene)

every 3D object which can interact with camera rays must have. Therefore the class *RenderObject* is defined which inherits from *Transform* class and provides an abstract function *hit(ray, t_min, t_max)*. That function later has to be implemented by the sub classes of *RenderObject* like *Sphere*.
In case of the *Sphere* class the intersection of ray and sphere is calculated using an optimized equation based on the radius and position of the sphere. Because most of the times two intersections will be detected for each ray, only the closer point is used. The function also returns the normal in the intersection point and flips it if the ray was emitted from within the sphere. On this place the implementation varies from the approach in [1]. The ray is outside the sphere if the dot product of the ray's direction and the normal in the intersection point is greater than or equal to 0 (if both are orthogonal it's still outside), while there is only $<$ used in [1]. For numerical approaches this makes no big difference, because this case rarely occurs but for correctness and understanding this was changed (logical difference between Listing 17 and Listing 18 in [1]).
The *hit(ray, t_min, t_max)* also allows to limit the render distance by the t parameters (close and far range) and returns color (or later material) of the sphere. Adding five spheres of different radius and color to a 3D scene of *Scene* class (described in section 4.1) results in a rendered image like in fig. 3.

4.1 Scene

A 3D scene can contain a bunch of different objects which are rendered from one or more cameras. The *Scene* class represents such a scene and allows to handle one or more cameras of *Camera* class which render objects of *RenderObject* class within the scene by calling the *Scene* objects *render()* function which is successively calling the cameras *render()* functions.
This class is the interface for the user to generate the desired scene and render it within just on python file like for *scene3.py* in listing 2.

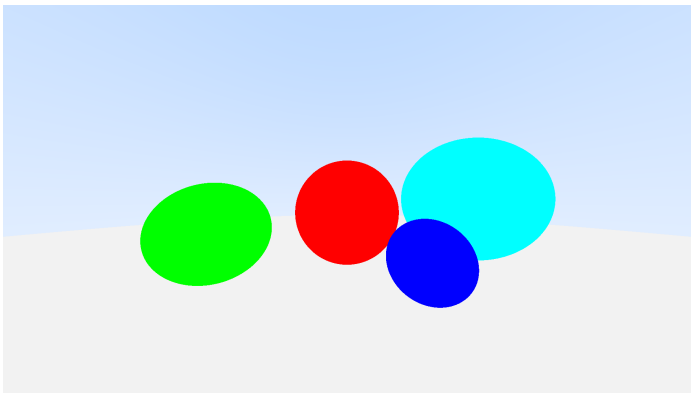

Figure 3: Scene of 5 spheres with different plain colors

Listing 2: Example for Scene class usage from scene3.py

```
from geometries import Vector
from objects import Scene, Camera, Sphere

scene = Scene("../images/image3")
# camera
main_camera = Camera(16 / 9, 1920, 1)

# spheres
s0 = Sphere(Vector(0, −1, −13), 4, color=Vector(1, 0, 0))
s1 = Sphere(Vector(−8, −2, −10), 3, color=Vector(0, 1, 0))
s2 = Sphere(Vector(4, −3, −8), 2, color=Vector(0, 0, 1))
s3 = Sphere(Vector(10, 0, −14), 5, color=Vector(0, 1, 1))
s4 = Sphere(Vector(0, −1005, −2), 1000, color=Vector(.9, .9, .9))

scene.add_render_object(s0)
scene.add_render_object(s1)
scene.add_render_object(s2)
scene.add_render_object(s3)
scene.add_render_object(s4)

scene.add_cam(main_camera)
scene.render()
```

References

[1] Peter Shirley. Ray tracing in one weekend, December 2020. https://raytracing.github.io/books/RayTracingInOneWeekend.html.

[2] Peter Shirley. Ray tracing: The next week, December 2020.