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 [?] and its continuation Ray Tracing: The Next Week [?]. 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 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

```python
image = Image(1920, 1080)
for y in range(image.height)[::-1]:
        for x in range(image.width):
                image.image_list[y, x] = Vector(128, 64, 255).to_int_array()

image.save_image("../images/image1.ppm")
```

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

3 Static camera

4 Spheres

5 Antialiasing

6 Materials

6.1 Diffuse

6.2 Specular

6.3 Specular transmission

6.4 Emissive

7 Enhancing camera

7.1 Positioning and orienting

7.2 Depth of field

8 Conclusion