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 ~\cite{} and its continuation Ray Tracing: The Next Week \cite{}. 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.

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).

Code scene 1

Image scene 1

Camera

The implementation of the camera in the Camera class largely follows the tutorial in ~\cite{}. 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 ~\cite{} 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.

Spheres

Just like the implementation of the camera the implementation of spheres within the Sphere class fast straight forward understanding the concepts from \cite and transfer them to Python. Key feature is the \emph{hit(ray, t\_min, t\_max)} function every 3D object which interacts wit camera rays must have. Therefore the class RenderObject is defined which inherits from Transform class and provides an abstract function \emph{hit(ray, t\_min, t\_max)}. That function later has to be implemented by the subclasses of RenderObject like Sphere.

In case of the Sphere class the intersection of ray and sphere is calculated using 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 maybe flips it if the ray was emitted from within the sphere. Here the implementation varies from the approach in \cite. The ray is outside the sphere if the dot product of ray’s direction and normal on spheres surface is greater than or equal to 0 (if both are orthogonal it’s still outside), while there is only $<$ used in \cite. 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 \cite).

The \emph{hit(ray, t\_min, t\_max)} also allows to limit the render distance (close and far range) by the \emph{t} parameters 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 ) result in a rendered image like in \

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 \cref.

The main difference between this implementation and the approach in ~\cite{} is a design decision regarding the rendering loop and the sent out rays.