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.

The main difference between this implementation and the approach in \cite{} is a design decision regarding the rendering loop and the sent out rays. To get such a ray, which can collide with objects later, the \emph{get\_ray(x, y, antialiasing)} function was implemented. While the tutorial uses the image plane coordinates for x and y, this implementation considers the pixel coordinates as x and y and transforms them into world space (image plane) coordinates in the function itself. This approach helps understanding, that for each image pixel one ray is generated. In case of antialiasing multiple rays for each pixel are sent out (antialiasing parameter) as explained more detailed in \cref{}. \cref{} shows the code of the \emph{get\_ray(x, y, antialiasing)} function.

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.

Antialiasing

Looking at the silhouettes of the spheres closely reveals some hard edges. The color of the background and the spheres can be clearly separated and do not fade into each other. Therefore, the final image seems less realistic and more pixelated. In order to get smoother transmissions antialiasing is used. In this case like in the tutorial \ac{msaa} is implemented.

Doing this, instead of generating just on ray sent out per image pixel as in \cref{}, multiple of those rays are generated. To obtain a small direction offset for each ray the intersection position with the image plane (ray goes through camera origin and this point) is randomly shifted.

This random factors for \emph{x} and \emph{y} are calculated using \emph{NumPy’s random.uniform(0, 1)} function, which returns a random value in the interval [0, 1). Without this offset the ray would go through the lower-left corner of each pixel in the image plane. This calculations are done in \cref{}, all additional lines of code regarding lenses are explained in \cref{}.

The color information of the intersections of those rays are summed up per pixel and later divided by the amount of samples per pixel (given to the constructor of an object of \emph{Camera} class by samples\textunderscore per\textunderscore pixel parameter).

The \emph{clamp(min, max)} function implemented in the tutorial to clamp the colors between 0 and 255 (probably due to a C++ limitation or personal favor), is not used in this implementation, because all single colors are used to be between 0 and 1. The sum of all colors therefore can’t be negative nor bigger than $255\cdot samples\textunderscore per\textunderscore$. The transformation to an rgb value between 0 and 255 is done in the \emph{write\textunderscore color(pixel\textunderscore color)} function together with gamma correction later on.

Materials

The following four subsections will introduce different materials. All of those materials are different variations of the same concepts. The idea is to basically abstract a surface material by two functions:

\emph{scatter(self, ray, pos, norm, front\_face)} method using the intersection information to return the color of the material used as attenuation and a ray which is again sent out from the intersection point

\emph{emit()} method to simulate light emitted by the object

Because all implemented objects just override the \emph{scatter} and/or the \emph{emit} method a super class \emph{Material} is implemented and all classes representing materials inherit it.

Finally in ray\_color on an intersection both methods are called on the intersected material. For the scattered ray ray\_color is called recursively and in each step attenuation and returned color are multiplied elementwise. The recursion depth is restricted by \emph{max\textunderscore bounce\textunderscore depth parameter given to camera constructor on camera initialization. The emitted light is added in each step resulting in \cref() for each recursion step.

Diffuse

For a diffuse material just \emph{scatter} method must be overwritten. The implementation follows the tutorial \cite{}. Therefore, an attenuation color and a scattered ray have to be returned:

The attenuation color is directly defined for a diffuse material as it’s \emph{albedo} color.

The scattered ray is sent out in direction of the normal in the intersection point plus a random offset vector. This offset vector is a unit vector in a unit sphere. Basic concept of this procedure is to simulate true Lambertian reflection.

The corresponding code is shown in \cref{}

Specular

Specular material is implemented similar to diffuse material. It also uses an albedo color for attenuation color. Main difference is the returned, scattered ray. On specular surfaces light is not randomly scattered but reflected. This is covered by law of reflection as described in lecture and tutorial and implemented in \emph{}