# Ray-tracing based renderer from scratch in Python

Jörn Lasse Vaupel

February 24, 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 functionalities are just easier to implement 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

```
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 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 main difference between this implementation and the approach in [1] 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 *get_ray(x, y, antialiasing)* method 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 method 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 section 5. Listing 5 shows the code of the *get_ray(x, y, antialiasing)* method.
The resulting image generated with *scene2.py* 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 $\phi$ in [1] results from focal length $fl = 1$, viewport height $vp_h = 2$ and aspect ratio $ar = 16/9$ as shown in eq. (1).

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

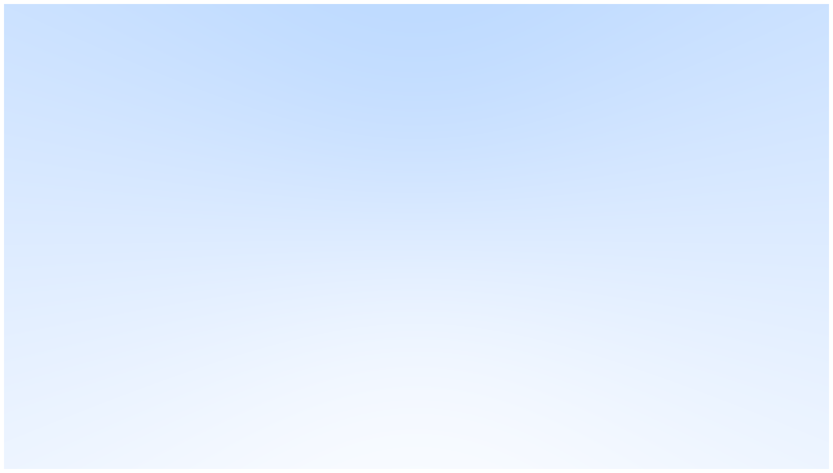
Figure 2: Gradient image generated by camera (empty 3D scene)

## 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)* method 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 method *hit(ray, t_min, t_max).* That method 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 method 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 is used by *ray_color(ray, scene, depth)* method to return 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.

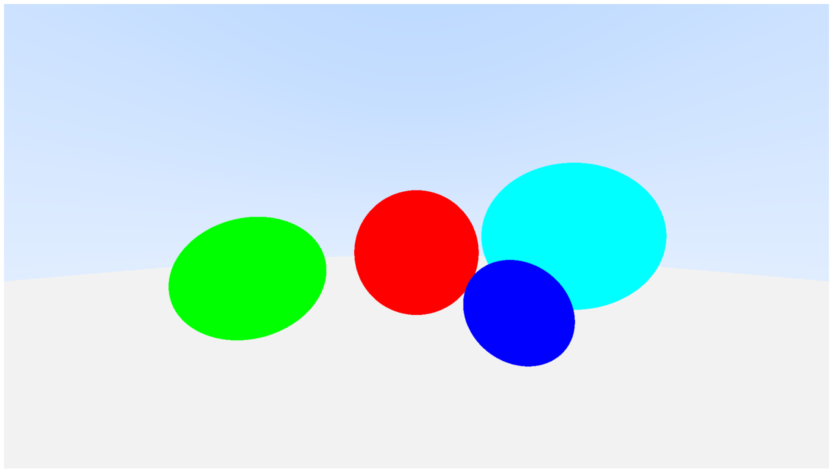

Figure 3: Scene of five spheres with different plain colors

## 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()* method which is successively calling the cameras *render()* methods.
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.

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

```python
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()
```

## 5 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 Multi Sampling Anti Aliasing (MSAA) is implemented. Doing this, instead of generating just on ray sent out per image pixel as in section 3, 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 $x$ and $y$ are calculated using *NumPy's random_uniform(0, 1)* method, 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 listing 5, all additional lines of code regarding lenses are explained in section 7.2.
The color information of the intersections of those rays returned by *ray_color(ray, scene, depth)* method are summed up per pixel and later divided by the amount of samples per pixel (given to the constructor of an object of *Camera* class by samples_per_pixel parameter).
The *clamp(min, max)* method 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 $255 \cdot samples\_per\_pixel$. The transformation to RGB values between 0 and 255 is done in the *write_color(pixel_color)* method together with gamma correction later on.
Based on this implementation images with different levels of MSAA can be calculated with *scene4.py* (MSAA-1 is like without antialiasing). A detailed comparison is visualized in fig. 4. While MSAA-2 is still quite pixelated, MSAA-16 looks clearly smoother. Following images will be rendered with MSAA-64, to avoid aliasing as a source of error.

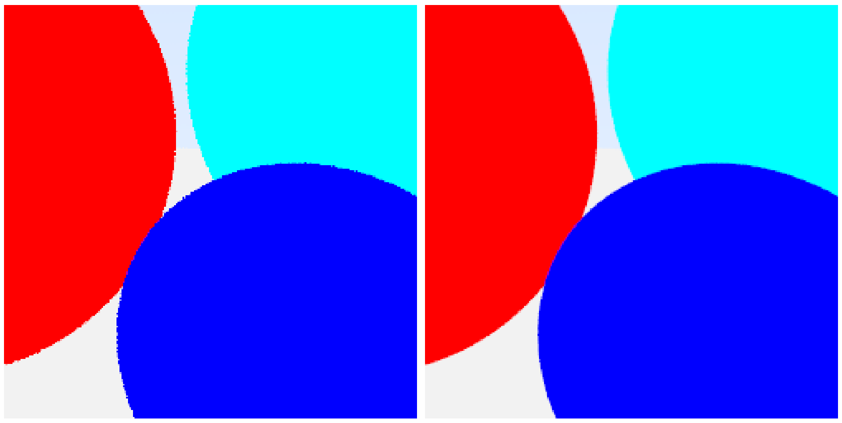

Figure 4: Cropped images of spheres with MSAA-2 (left) and MSAA-16 (right)

## 6 Materials

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

1. *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

2. *emit()* method to simulate light emitted by the object

Because all implemented objects just override the *scatter* and/or the *emit* method a super class *Material* is implemented and all classes representing materials inherit it.
Finally, in *ray_color(ray, scene, depth)* method both methods are called on an intersected material. For the scattered ray *ray_color* is called recursively and in each step *attenuation* and returned color ($color_{i+1}$) are multiplied element-wise and the depth parameter is decreased by 1. The maximal recursion depth is restricted by *max_bounce_depth* parameter given to camera constructor on camera initialization. The emitted light ($emitted\_color$) is added in each step resulting in eq. (2) for each recursion step $i$.

$$color_i = emitted\_color + color_{i+1} \cdot attenuation \tag{2}$$

Due to this implementation for a *max_bounce_depth* of 1 all objects (intersected by a ray) will occur black (if they are not emissive). This is the reason, why the first image for diffuse (*scene5.py*), specular (*scene6.py*) and specular transmissive (*scene7.py*) objects generated in the corresponding Python module looks as shown in fig. 5.

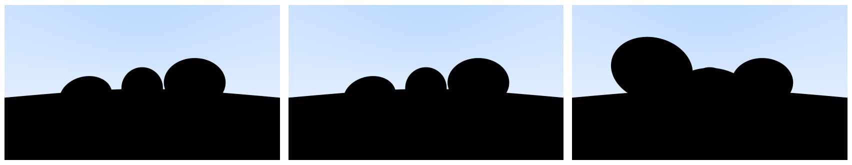

Figure 5: Example images resulting from a *max_bounce_depth* of 1

In order no produce realistic renderings *gamma correction* and a fix for *shadow acne* are implemented in *Camera* class in this step (following the tutorial).

## 6.1 Diffuse

For a diffuse material just *scatter* method must be overwritten. The implementation follows the tutorial [1]. Therefore, an attenuation color and a scattered ray have to be returned:

1. The attenuation color is directly defined for a diffuse material as it's *albedo* color.
2. 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 of the *DiffuseMaterial* class representing diffuse materials is shown in listing 3.

Listing 3: *DiffuseMaterial* class

```
class DiffuseMaterial(Material):
    # implementation of a diffuse material (inherits from Material)
    def __init__(self, albedo):
        self.albedo = albedo

    def scatter(self, ray, pos, norm, front_face):
        # returns scattered ray and color of material
        scatter_dir = norm + Vector.rand_in_unit_sphere().normalize()
        if scatter_dir.near_zero():
            scatter_dir = norm
        return Ray(pos, scatter_dir), self.albedo
```

Applying diffuse materials in *scene5.py* on the 5 spheres shown in fig. 3 results in fig. 6.

## 6.2 Specular

Specular material implemented in *SpecularMaterial* class is similar to diffuse material. It also uses an albedo color for attenuation color. Main difference is

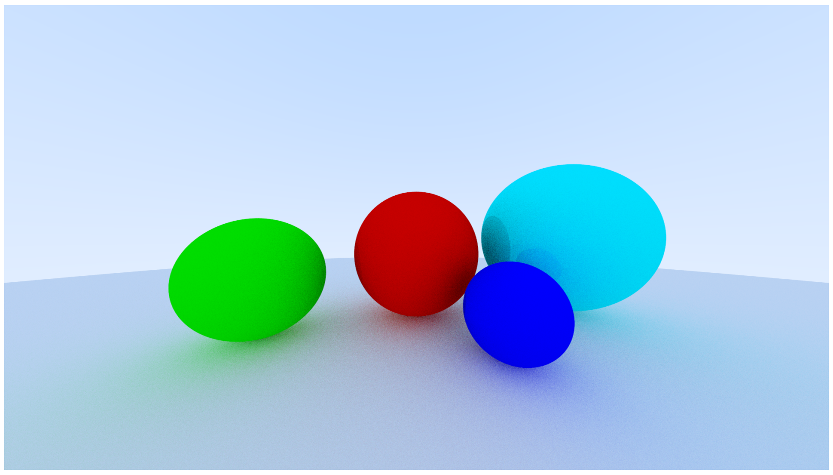

Figure 6: Scene of five spheres with diffuse material (*max_bounce_depth* of 16)

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 *reflect(self, norm)* (static) function called on the direction vector of the ray with the normal vector as the function parameter.

Listing 4: Function reflecting a vector on a normal vector

```
def reflect(self, norm):
    return self - norm * (self * norm) * 2
```

By adding 3 spheres to the last scenario in *scene6.py* and applying different specular materials on them, the image in fig. 7 can be generated. One of those spheres is glossy as described in a bit more detail in section 6.3.

## 6.3 Glossy

In order to make a specular surface appear glossy a random offset is added to the direction of the reflected ray. This random offset is a vector within an unit sphere which is scaled by the so called *fuzz* factor between 0 and 1. This factor is given to the material class by its constructor. This results in the default specular material for a *fuzz* factor of 0 and more glossy specular materials for higher factors. If one of those scattered rays now do not point out of the sphere (no acute angle between normal and ray), the color black is returned (no light) and the recursion of *ray_color* is stopped.

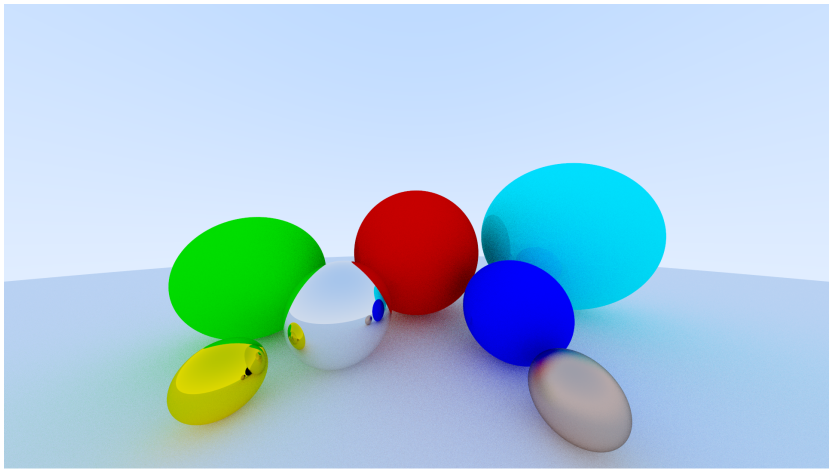

Figure 7: Scene of several spheres with diffuse and specular material (*max_bounce_depth* of 16)

### 6.4 Specular transmission

### 6.5 Emissive

## 7 Enhancing camera

### 7.1 Positioning and orienting

### 7.2 Depth of field

Listing 5: Method to generate a ray for given pixel coordinates using antialisaing and thin lens approximation

```
def get_ray(self, x, y, antialiasing):
    # antialiasing offset
    rand_offset_x, rand_offset_y = 0, 0
    # depth of field offset (lens)
    lens_offset = Vector.rand_in_unit_disc() * self.lens_radius
    position_offset = self.u * lens_offset.x + self.v * lens_offset.y

    if antialiasing:
        rand_offset_x = np.random.uniform(0, 1)
        rand_offset_y = np.random.uniform(0, 1)

    return Ray(self.position + position_offset, self.lower_left_corner
        + self.horizontal * (x + rand_offset_x) / (self.image_width - 1)
        + self.vertical * (y + rand_offset_y) / (self.image_height - 1)
        - self.position - position_offset)
```

## 8 Conclusion

## References

## References

[1] Peter Shirley. *Ray Tracing in One Weekend*. Dec. 2020. URL: `https://raytracing.github.io/books/RayTracingInOneWeekend.html`.

[2] Peter Shirley. *Ray Tracing: The Next Week*. Dec. 2020. URL: `https://raytracing.github.io/books/RayTracingTheNextWeek.html`.