Ray-tracing based renderer from scratch - an implementation in Python -

Jörn Lasse Vaupel

February 27, 2023

Abstract

This report describes the realization of a custom ray-tracing based renderer in Python following a web tutorial and discussing similarities and differences between both implementations.

Contents

1 Introduction **3**

2 First rendering loop **3**

3 Camera **4**

4 Spheres **5**
4.1 Scene 6

5 Antialiasing **7**

6 Materials **8**
6.1 Diffuse 9
6.2 Specular 10
6.2.1 Glossy 10
6.3 Specular transmissive 11
6.4 Emissive 12

7 Enhancing camera **12**
7.1 Field of View (FOV) 13
7.2 Positioning and orientation 13
7.3 Depth of field 14

8 Performance and further steps **15**

9 Conclusion **15**

References **16**

1 Introduction

This report summarizes the process of implementing a ray-tracing based renderer from scratch following the Peter Shirley's introduction *Ray Tracing in One Weekend* [4] and its continuation *Ray Tracing: The Next Week* [5] (in the folowing both are referred as tutorials). 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, which would change the static color slightly. This shows that some design decisions in this implementation like gamma correction or a horizontal Field of View (FOV), 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 buffered in an image array maintained by an *Image* class before it is saved in a .ppm file (defined in [3]). 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 [4]. In general, it's necessary to define a large set of algebraic methods within the *Vector* and *Ray* class. Regarding 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 [4] 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 the tutorial 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 is 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 6 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 precise - due to personal favor, the FOV is slightly different in all rendered images (more

details in section 7.1).

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 [4] 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 offer. 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. Here the implementation varies from the approach in [4]. 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 [4]. 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 [4]).
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.

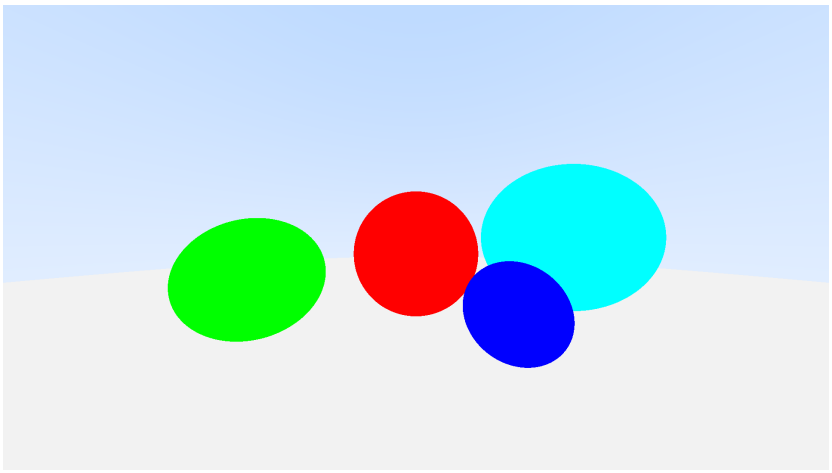

Figure 3: Scene of five spheres with different plain colors

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. That method is successively calling the *render()* method of each single camera of the scene. *Scene* class is the interface for the user to generate the desired scene and render it within just on Python module like for *scene3.py* in listing 2.

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

```
1  from geometries import Vector
2  from objects import Scene, Camera, Sphere
3
4  scene = Scene("../images/image3")
5  # camera
6  main_camera = Camera(16 / 9, 1920, 1)
7
8  # spheres
9  s0 = Sphere(Vector(0, −1, −13), 4, color=Vector(1, 0, 0))
10 s1 = Sphere(Vector(−8, −2, −10), 3, color=Vector(0, 1, 0))
11 s2 = Sphere(Vector(4, −3, −8), 2, color=Vector(0, 0, 1))
12 s3 = Sphere(Vector(10, 0, −14), 5, color=Vector(0, 1, 1))
13 s4 = Sphere(Vector(0, −1005, −2), 1000, color=Vector(.9, .9, .9))
14
15 scene.add_render_object(s0)
16 scene.add_render_object(s1)
17 scene.add_render_object(s2)
18 scene.add_render_object(s3)
19 scene.add_render_object(s4)
20
21 scene.add_cam(main_camera)
22 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 is randomly shifted (ray goes through camera origin and this point) .
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 6, all additional lines of code regarding lenses are explained in section 7.3.
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, which is 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 (internal color). The sum of all colors therefore can't be negative nor bigger than *samples_per_pixel* value. The transformation to RGB values between 0 and 255 (external color) is done in the *write_color(pixel_color)* method together with gamma correction later on.

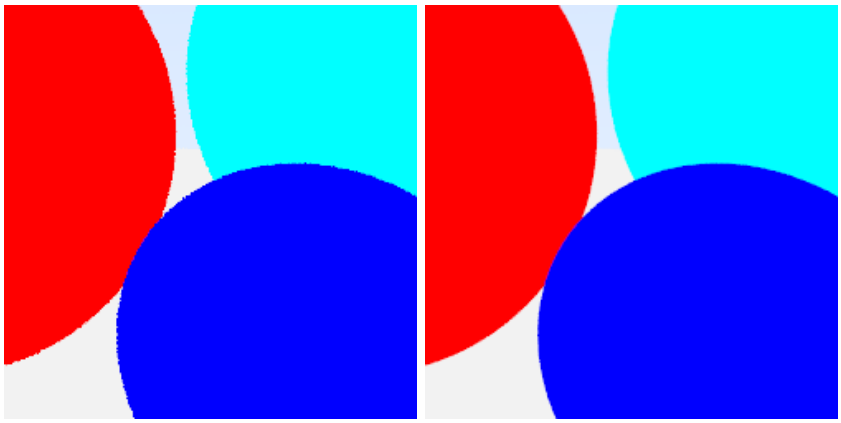

Figure 4: Cropped images of spheres with MSAA-2 (left) and MSAA-16 (right)

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.

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 (default black color)

Because all implemented materials 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. (1) for each recursion step i.

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

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.

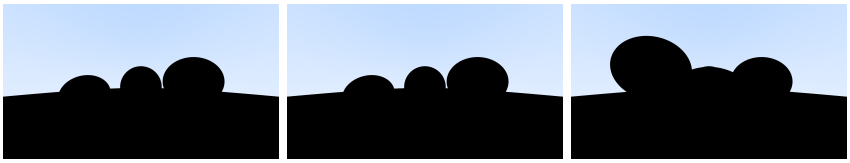

Figure 5: Example images resulting from a *max_bounce_depth* of 1

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 [4]. 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

```
1  class DiffuseMaterial(Material):
2      # implementation of a diffuse material (inherits from Material)
3      def __init__(self, albedo):
4          self.albedo = albedo
5
6      def scatter(self, ray, pos, norm, front_face):
7          # returns scattered ray and color of material
8          scatter_dir = norm + Vector.rand_in_unit_sphere().normalize()
9          if scatter_dir.near_zero():
10             scatter_dir = norm
11         return Ray(pos, scatter_dir), self.albedo
```

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

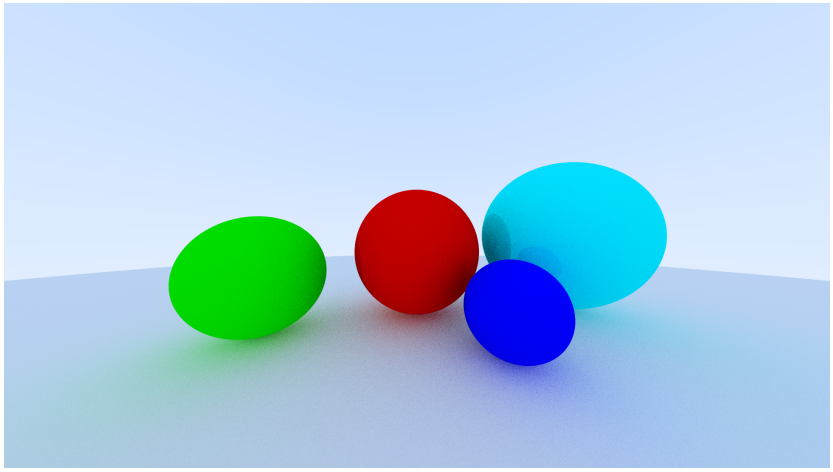

Figure 6: Scene of five spheres with diffuse material (*max_bounce_depth* of 16)

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 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 the tutorial and implemented in *reflect(self, norm)* method shown in listing 4 called on the direction vector of the ray with the normal vector as the function parameter.

Listing 4: Method for reflecting a vector on a normal vector

```
1 def reflect(self, norm):
2     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 6.2.1.

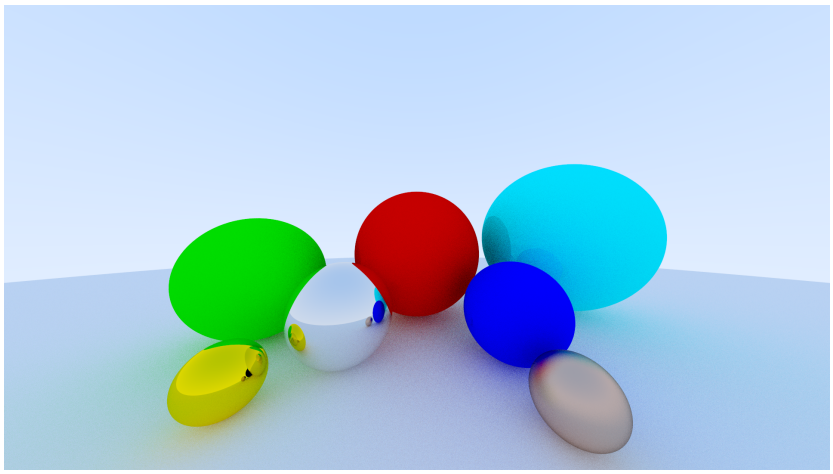

Figure 7: Scene of several spheres with diffuse and specular material (*max_bounce_depth* of 16)

6.2.1 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.

6.3 Specular transmissive

For specular transmissive materials in addition to reflection the laws of refraction (Snell's law) must be taken into account for the calculation of the scattered rays direction and are therefore implemented as shown in listing 5. Basic concept is to use an Index Of Reflection (IOR) which is given to the constructor of the *TransmissiveMaterial* class (no albedo color is used in this implementation). This IOR is a material specific constant and the reciprocal of the refraction ratio (list of values in [1]). With that information it's decided if reflection or refraction have to be used to determine the scattered ray's direction. In addition, as explained in the tutorial in [4] Schlick's approximation is applied to get more realistic results.

Listing 5: Method for refracting a vector on a normal vector

```
1 def refract(self, norm, cos_theta, eta_ratio):
2     # cos_theta: already calculated cosinus of angle between vector and normal
3     # eta_ratio: refraction ratio
4     r_out_perp = (self + (norm * cos_theta)) * eta_ratio
5     r_out_parallel = norm * -np.sqrt(np.abs(1-(r_out_perp*r_out_perp)))
6     return r_out_perp + r_out_parallel
```

With specular transmissive materials for example a glass ball can be simulated. But also hollow glass balls can be rendered easily by putting one sphere of smaller and negative radius within another. Such an hollow object and an additional transmissive sphere are added to the scene shown in the last section resulting in *scene7.py* module and the image in fig. 8. Because the hollow glass sphere is directly in front of the camera some interesting effects can be observed. For example some reflections within the sphere and distortions of objects, which are visible through and next to the sphere.

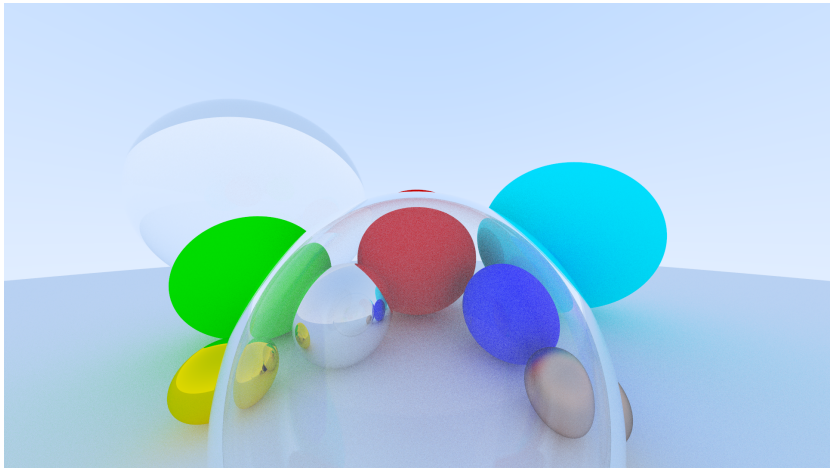

Figure 8: Scene of several spheres with diffuse, specular and specular transmissive material (*max_bounce_depth* of 16)

6.4 Emissive

Finally, in this implementation emissive materials are the only materials overwriting *Material* class's *emit()* method and also do not implement any kind of scattering. Main idea based on the second tutorial in [5] is, to use a light *color* and an *intensity* given to the constructor of *EmissiveMaterial* class and return the *color* scaled with the *intensity* in the *emit()* method. This causes static color and therefore an unrealistic appearance of an emissive object itself, but it's an easy and valid approach for physical lights giving good results .
While emissive objects also influence scenes with global illumination, their effects are quite more visible if no background lightning is used. Therefore, the *Camera* class is extended by adding the possibility to change the color (gradient) of the background. By setting it to black and placing three spheres with emissive materials of different *color* and an *intensity* in *scene8.py* the image in fig. 9 can be rendered. In order to generate high quality results and realistic lighting in addition to *samples_per_pixel* also *max_bounce_depth* is set to 64.

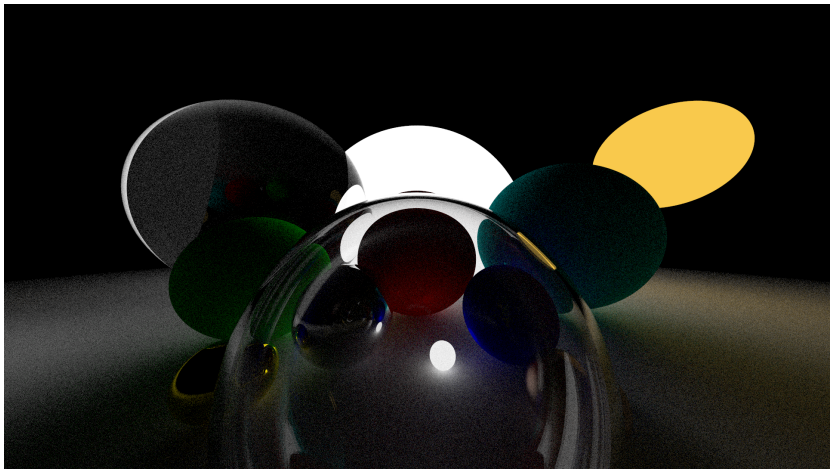

Figure 9: Scene of several spheres with diffuse, specular, specular transmissive and emissive material without global illumination

7 Enhancing camera

The *Camera* class introduced in section 3 provides already good results, but in order to get more realistic renderings three additional aspects are implemented as described in the following three subsections.

7.1 FOV

To this point vertical and horizontal FOV are indirectly defined by the viewport size and the focal length parameter of the camera as shown in the tutorial. By introducing a parameter for vertical FOV as in [4] or for horizontal FOV as in this implementation (due to personal favor; like in most video games) the viewport size results from FOV and focal length.
It's important to mention that for all rendered images the default horizontal FOV is set to 130° (representing human horizontal FOV), while the FOV ϕ in [4] results from focal length $fl = 1$, viewport height $vp_h = 2$ and aspect ratio $ar = 16/9$ as shown in eq. (2).

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

Setting focal length always to one as in [4] is not a problem because all calculations are correctly implemented for this specific case, but for different focal length values wrong equations would be used to calculate the viewport sizes. Therefore, if focal length as well as FOV are parameters of *Camera* class, both values have to be used for correct viewport calculations with rearranged eq. (2).
Additionally, the trick for thin lens approximation explained in section 7.3 is easier to implement, if focal length is used in all equations on default.

7.2 Positioning and orientation

A main feature of real cameras is that they can be moved around and the camera's angle can be changed. In order to provide this feature also with *Camera* class the approach from the tutorial is used.
A *lookfrom* vector defining the camera's position, a *lookat* vector defining the point in space the camera is looking to and a up vector defining the up direction in camera's own coordinate system are given to the constructor of *Camera* class. Using those three vectors an orthonormal basis of camera specific coordinate system can be calculated using cross products. This coordinate system is now used instead of forward, right and up vectors in world space coordinates.
By adding five more spheres to the current scene and changing camera's position and orientation in *scene9.py* it's possible to render images from interesting perspectives as shown in fig. 10.

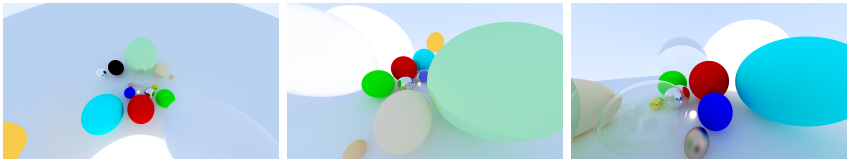

Figure 10: Scene of several spheres with various materials rendered from different perspectives

7.3 Depth of field

Another effect in real photography is depth of field. Due to thin lenses and apertures used in real cameras only objects in a limited range around a certain distance are perfectly sharp others appear out of focus. While it's also an advantage of rendering to be able to generate globally sharp images, often depth of field is a desired effect.
There are different ways to simulate this effect. For example a physical approach can be used by calculating refraction effects of a lens of certain thickness and radius (scaled up to complex structures of multiple convex and/or concave lenses as in [2]).
The *Thin lens approximation* used in this implementation inspired by [4] avoids those complex and intense calculations by simulating the lens with a random offsets for the camera center and a clever usage of focal length.
Having a last closer look on listing 6 shows the implementation of random offsets of the camera's center. All camera rays are not longer sent out only from the center but from a random position within an unit disc of aperture radius around the camera center. Only objects which are located on the image plane will be rendered perfectly sharp as before, others will be blurred due to the slightly different ray direction in front of and behind the image plane. The diameter of the lens is given to the constructor of the *Camera* class as the *aperture* parameter, if it's set to 0 no defocus blur is simulated. Finally, the last step of this *Thin lens approximation* is to set focal length to the desired depth of field to grant the expected behavior (it's only that simple if focal length is already used in all equations as described in section 7.1).

Listing 6: 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)
```

The effect of a bigger value for *aperture* diameter resulting in a lower depth of field is shown in fig. 11 which is generated using *scene10.py*.

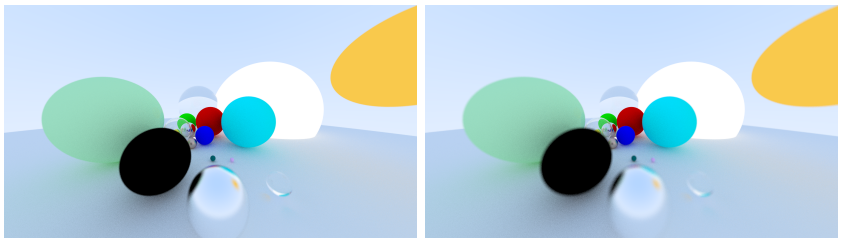

Figure 11: Scene of several spheres with various materials rendered with a *aperture* diameter of 0.2 units (left) and 0.5 units (right))

8 Performance and further steps

To pick up the considerations about performance discussed in section 1 it's important to mention, that the render time of an image scales strongly with the number of objects and different materials in the scene and the parameters given to the camera. For example an image with full hd resolution (1920x1080) rendered with MSAA-64 and a *max_bounce_depth* of 64 showing several spheres of different materials takes up to 10h or more. This is due to both Python and the missing efficiency improvements like parallelization.
Parallelization could be done on the CPU or as usually in modern computer graphic applications on the GPU. While GPU parallelization (for example with CUDA for Nvidia GPUs in Python) is difficult to implement and goes beyond the scope of this project, CPU parallelization can be achieved much easier. Especially if multiple images must be rendered the same time, it's easy to create two Python modules which define a scene and start both modules independently. Without more effort and complicate implementations the efficiency can be increased greatly.
Nevertheless, implementing GPU parallelization would be an interesting next step regarding the increase of performance.
In addition concepts described in [5] like other objects next to spheres and texture mapping would be good choices for further steps, because they are granting great versatility and allow many new use cases of the renderer.

9 Conclusion

Using Python was connected to the advantage of clean code and readability on the one hand and concerns about performance problems on the other hand. All in all, the implementation and the results shown in this report legitimize the decision for Python as a programming language for the renderer. Not due to it's performance as described in section 8 but the opportunity to understand the fundamental concepts of a ray-tracing based renderer in an educational context, which was the purpose of this project. It's features and readability granted to focus on the ideas of ray-tracing rather than the coding itself.

The tutorials ([4] and [5]) explain the basic concepts but also allow and motivate to dive deeper into equations and further concepts and the custom renderings from each step encourage to go further and further.

References

[1] *IOR LIST*. URL: https://pixelandpoly.com/ior.html.

[2] Matt Pharr, Wenzel Jakob, and Greg Humphreys. "Realistic Cameras". In: *Physically Based Rendering: From Theory To Implementation*. Oct. 2018. Chap. 6.4. URL: https://www.pbr-book.org/3ed-2018/Camera_Models/Realistic_Cameras.

[3] *ppm*. URL: https://netpbm.sourceforge.net/doc/ppm.html.

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

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

.