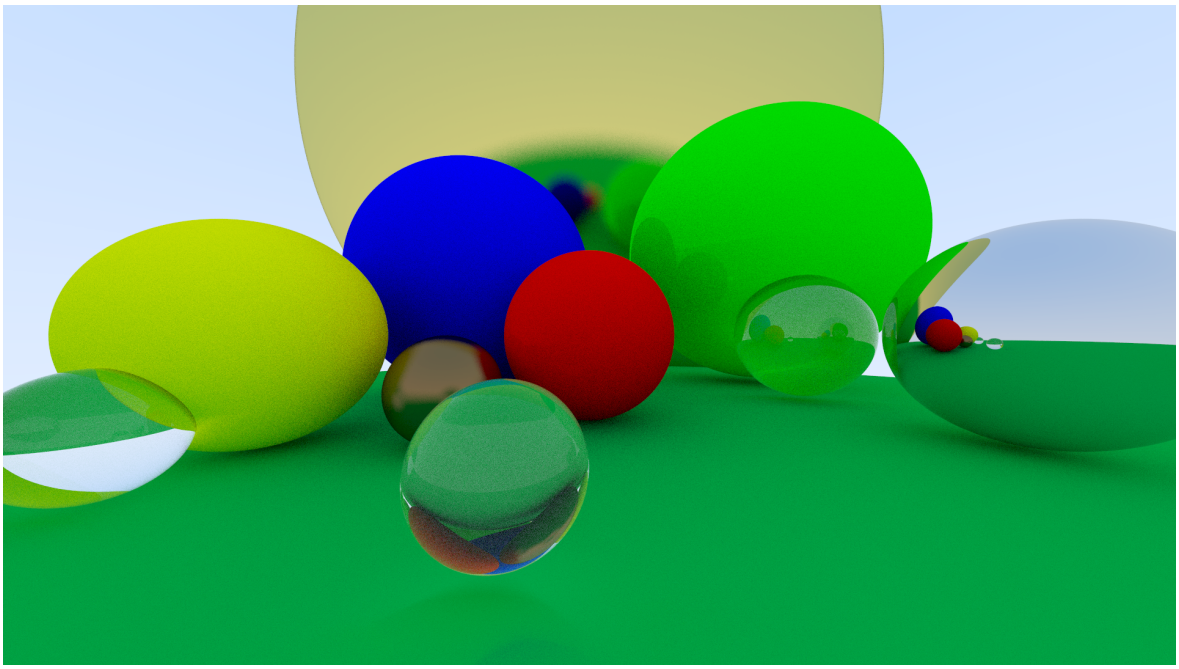
# Ray-tracing based renderer from scratch



Timo Salisch

# Contents

# 1 Rendering loop

The rendering process consists of two loops. One outer loop which iterates through the rows of the image and the inner loop iterating through the columns of the image. This way, every pixel will be visited once and the color of the pixel can be calculated. In this basic version of the rendering loop the color of every pixel will be set to (128, 64, 255). The resulting image can be seen in figure 1.1.

Figure 1.1: Result after running simple rendering loop

To store the already rendered information about the image, an *Image* class is implemented. An instance of this class represents one image and saves the color information of every pixel using a two dimensional list. The color is represented as an instance of the *Vector* class. This *Vector* class saves three floating point values and offers simple

arithmetic operations of (three-dimensional) vectors. These three values can either be interpreted as the world coordinates x, y and z or as the color properties red, green and blue. To change the color value of a specific pixel of the image, the *Image* class provides an *update* method to do so. In order for the image to be viewed, it has to be saved. The image is saved in the "PPM" format. This is done by the *save_image* method which expects a path and then saves the image to that path using the "PPM" format. For the file to be readable as "PPM", the first three lines have to include: "P3", image width and height, max color value. The actual pixel information are added by the loop iterating through the pixels. It is import to iterate through the rows beginning with the top row, otherwise the image is flipped. The saving process can be seen in source code 1.1.

**Source code 1.1:** Saving an image

```
1  def save_image(self, path: str):
2      image_str = f'P3\n{self.width} {self.height}\n255'
3
4      for j in range(self.height)[::-1]:
5          for i in range(self.width):
6              red, green, blue = self.matrix[i][j].to_tuple()
7              image_str = image_str + f"\n{int(red)} {int(
                   green)} {int(blue)}"
8
9      with open(path, mode='w+') as f:
10         f.write(image_str)
```

# 2 Camera

The project is extended by a *Ray* and *Camera* class. A ray can be used to find all objects that need to be projected onto one pixel and thus finding the color of a pixel. Every ray has an origin and a direction, with both being instances of the *Vector* class. It is possible to get the position of a ray through its method *position*. The camera represents the observer of the scene and has properties such as position and information about the image. The rendering loop from chapter 1 is moved into the camera class. It uses the cameras properties to find the color for every pixel. This is done by sending a ray from the camera to every pixel and finding the color of this ray. Because there are no objects in the scene yet, the color of a ray is the color of the background at that position as shown in source code 2.1.

**Source code 2.1:** Color of ray

```
1 def get_color(ray: Ray):
2     unit_direction = ray.direction.normalize()
3     t = 0.5 * (unit_direction.y + 1)
4     ray_color = Vector(255, 255, 255) * (1 - t) + Vector
        (127.5, 178.5, 255) * t
5     return ray_color
```

This results in an image which goes from light blue at the top to a white color at the bottom, as can be seen in figure 2.1.

Figure 2.1: Rendering with camera and colored background

# 3 Objects: shape

jhfjgjkfgjfghjjfg

# 4 Enhancing camera and rendering loop

jhfjgjkfgjfghjjfg

# 5 Objects material: diffuse

jhfjgjkfgjfghjjfg

# 6 Objects material: specular

jhfjgjkfgjfghjjfg

# 7 Objects material: specular transmission

jhfjgjkfgjfghjjfg

# 8 Lights

jhfjgjkfgjfghjjfg

# 9 Positioning and orienting camera

jhfjgjkfgjfghjjfg

# 10 Animation

jhfjgjkfgjfghjjfg

# 11 Render parallelization

jhfjgjkfgjfghjjfg

# 12 Config files

jhfjgjkfgjfghjjfg