Ray-tracing-based Renderer from Scratch

Task One: Rendering loop

For the first task, I created a struct in C++ to represent a single pixel in a PPM-image. This structure consists of the three RGB-Values and allows me to work with a fixed 2D-Array of pixels when storing image data in my code. The output image is calculated by providing a single color. The first example of a rendered image looked like this:

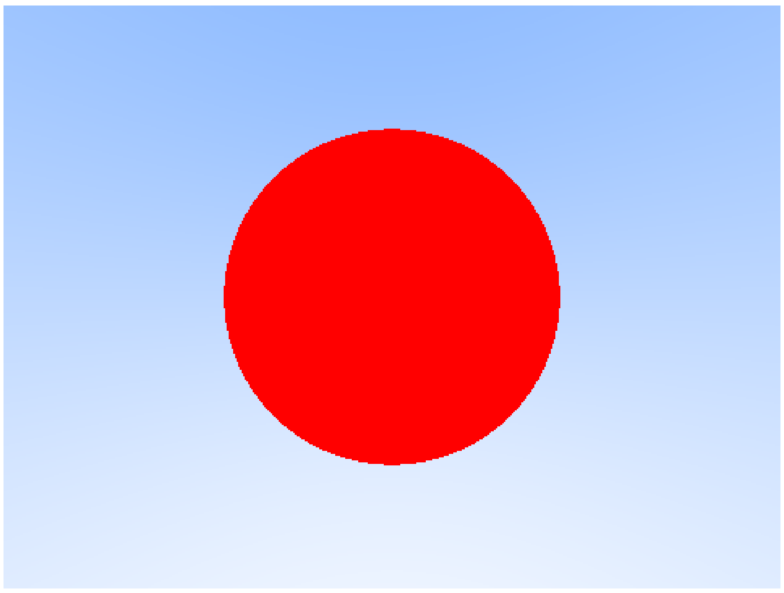
Task Two: Camera

In the second task, I created a camera class and set an instance of a camera to be in the origin. In the ray_color()-function, I created a background color which uses the height (y-value) to calculate the different colors. That's how we get a linear color gradient for the background.

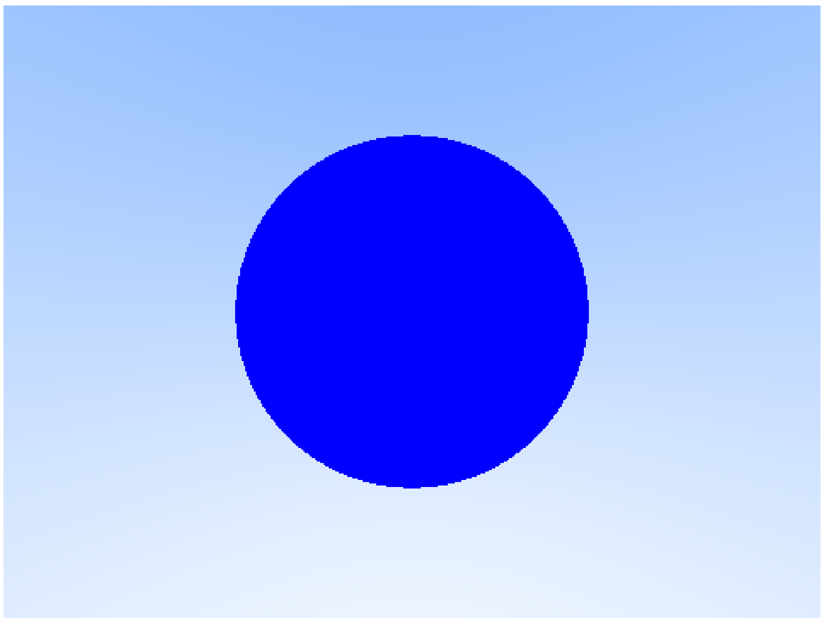
This gradient does not look linear, that's because we are in 3D Space and the camera is not facing perfectly frontward. I experimented with different values for aspect ratio and focal length and was convinced, that a focal length of "2.0" would be perfect. This did in fact not become clear until later on, when spheres were added and looked extremely distorted. Because I initially used a focal length of "1" or even less, an extreme fisheye effect occurred. The aspect ratio is 4:3, because I rendered the images in 640x480 resolution to have a middle ground between efficiency in rendering and a decent image quality.

Task Three: Objects and shape

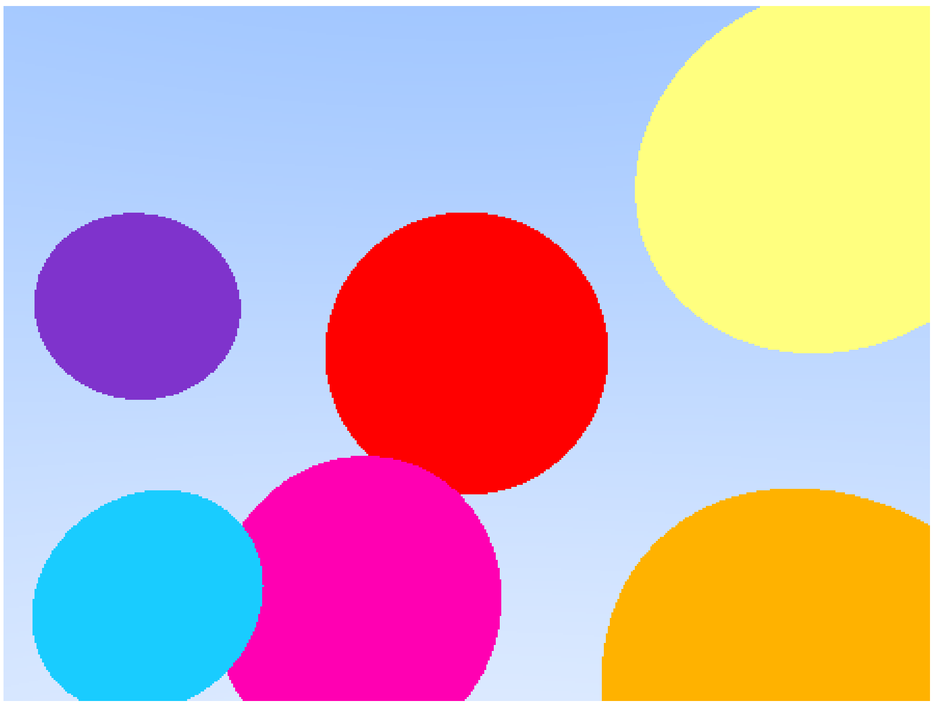
To store the intersection data, I created a struct called “hit” consisting of the values: “bool is_hit”, “vector intersection_point” “vector normal_intersection_point” and “color object_color”. If the “hit_sphere”-function of a sphere is called, it returns this struct. This method checks, whether the ray hits the sphere, if it does, the intersection point, corresponding normal vector and color are set. The first sphere I created just returned a solid color which was determined in the constructor of the sphere.



Because this only is a single sphere, I added more but quickly ran into the problem of overlapping spheres. In the following picture, there are supposed to be 2 spheres:

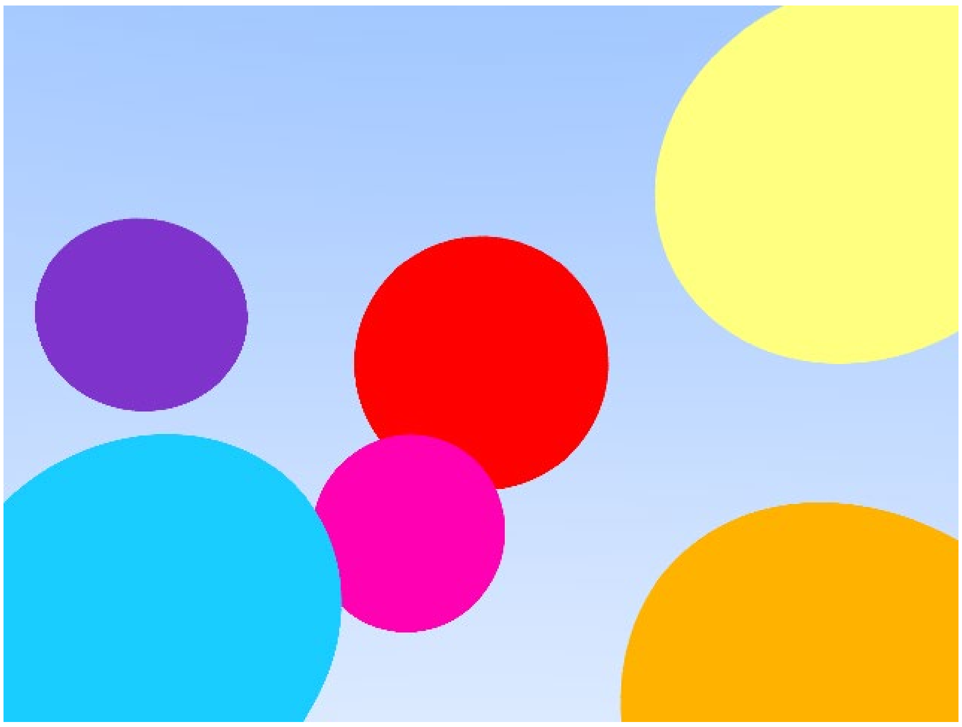


Unfortunately, the position relative to the camera is pretty hard to imagine if only the position vector and radius is given. But after a lot of trying, I managed to add 6 different spheres to the scene.



Task Four: Enhancing camera rays and rendering (AntiAliasing)

To achieve Anti-Aliasing, I used a random value from 0 to 1 and added it as an offset to the pixel. From this position, a color was calculated and the end result was multiplied by the inverse of the number of rays shot per pixel (effectively dividing by that number). The following example is using 16 rays per pixel, the other results (1,2,4,8,16) are also added in the folder.



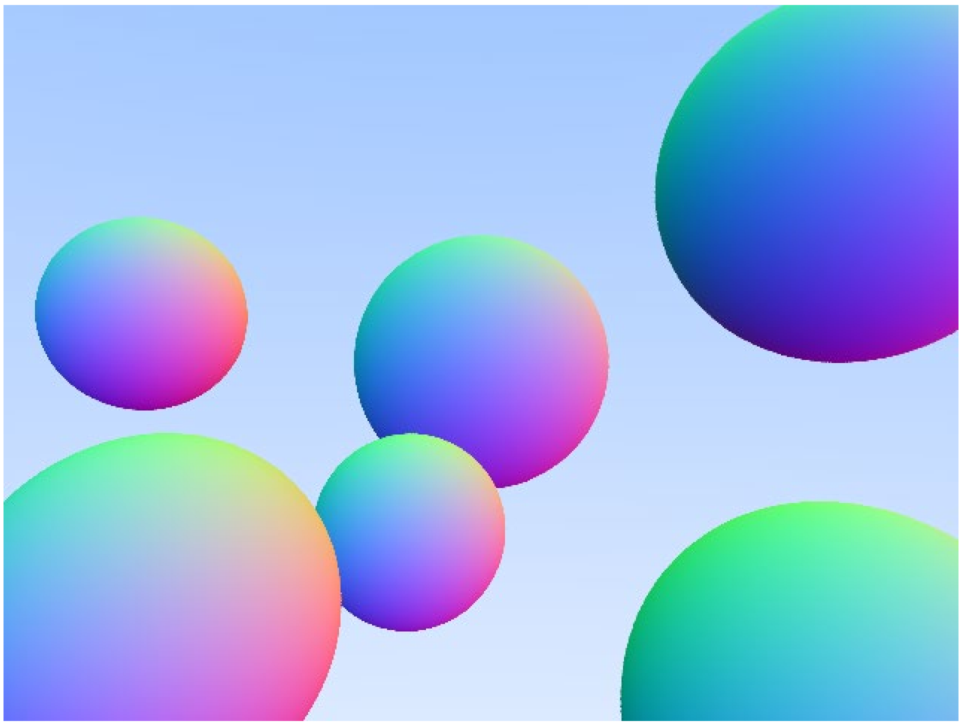
The jagged edges which can be seen in the first image are replaced by much smoother ones.

Task Five: object material: diffuse

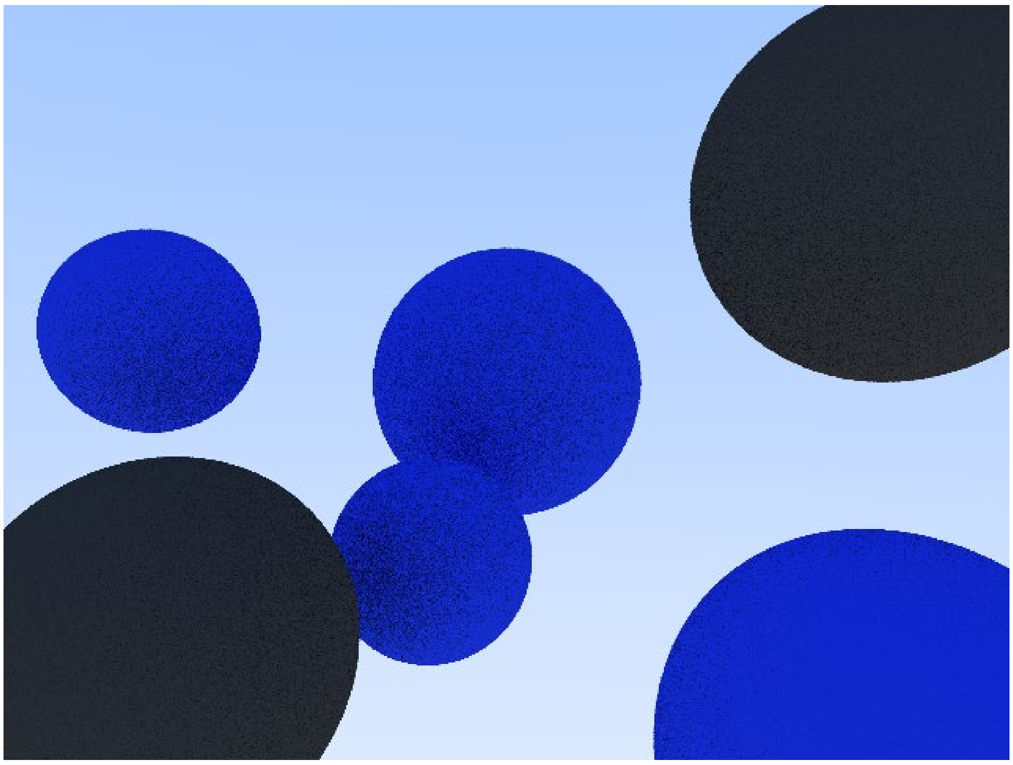
Before the diffuse material is created there is a simple method which can be used to test, if the model is working properly and the normal vectors in the intersection points are returned correctly. This is done by calculating the color of the sphere with the normal vector like this:

```
return 0.5 * (normal_intersection_point + color(1,1,1));
```

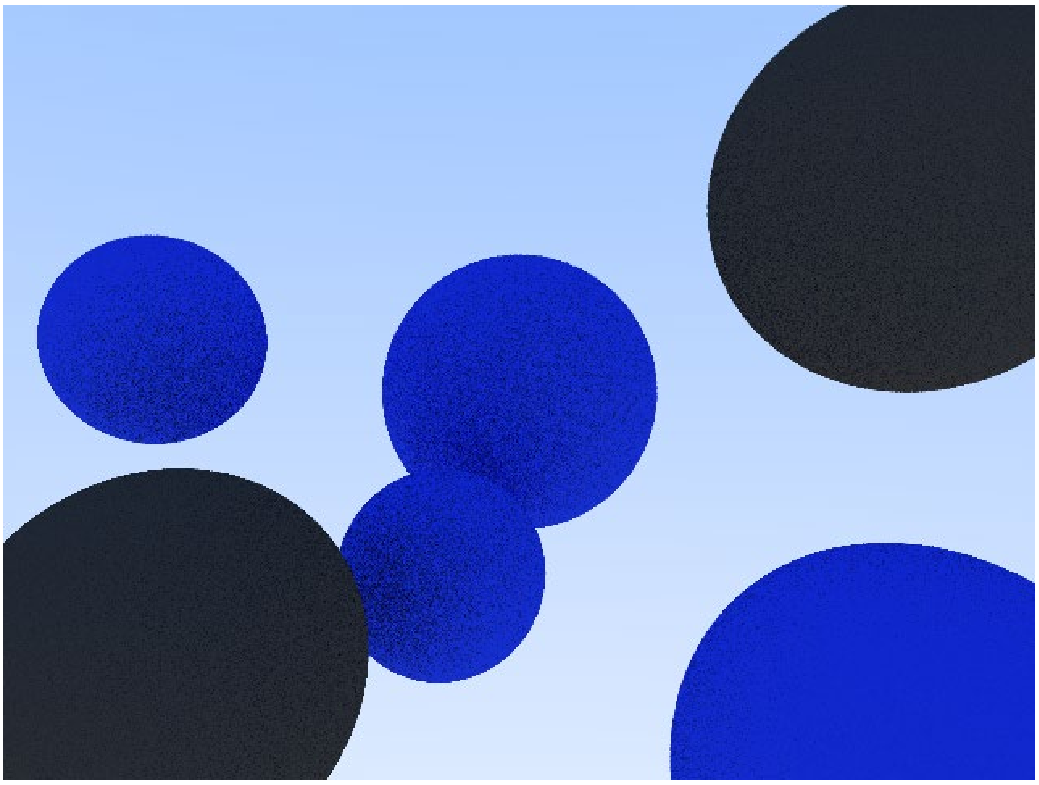
This returns the following image:



It becomes apparent that the model is working properly. The next step is to create the “diffuse” material class, which in my case is a subclass of “material”. This class accepts the albedo as a parameter and has a bool type variable “scatter”, which accepts pointers to a ray, the intersection point, a color and the next, scattered, ray. This bool sets the correct values for the output color and the scattered ray while using the inbound ray and the intersection point and returns “true” after that. It calculates the scattered ray by adding a random unit vector to the normal in the intersection point. To avoid so-called “shadow acne”, it is also checked if the generated random vector is extremely near to zero. If that’s the case, the normal vector is returned as the scattered ray. This bool is called by the ray_color()-function recursively. If this material is applied to every sphere in the image, the output image (recursion limit = 1) looks like this:

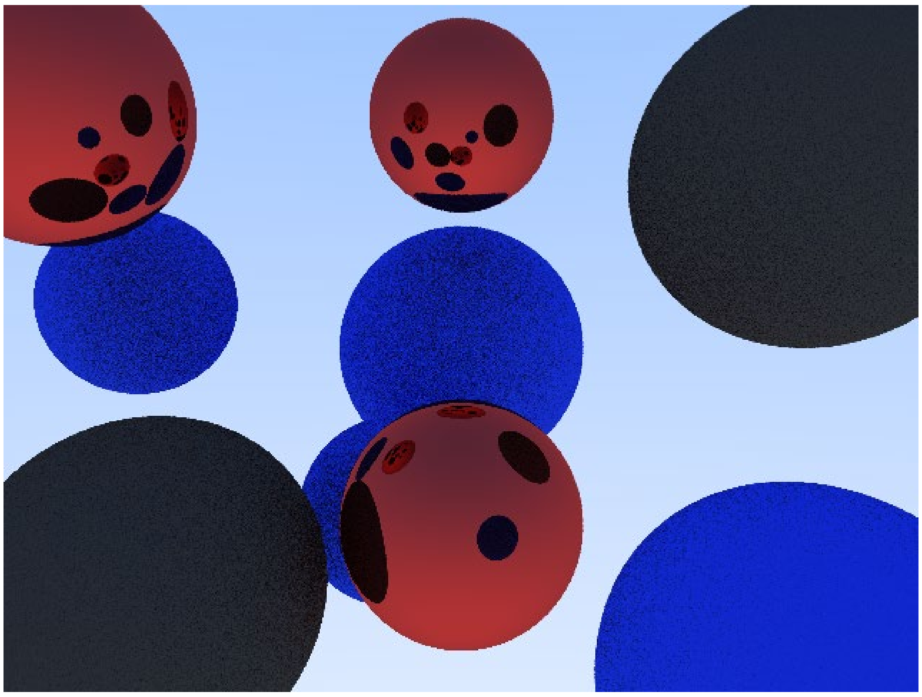


In this case, four spheres have the same material with a blueish albedo while the other two are dark grey. Unfortunately, playing with the recursion limit did not do a lot. The following example used the recursion limit of 16 and there is almost no recognizable difference.

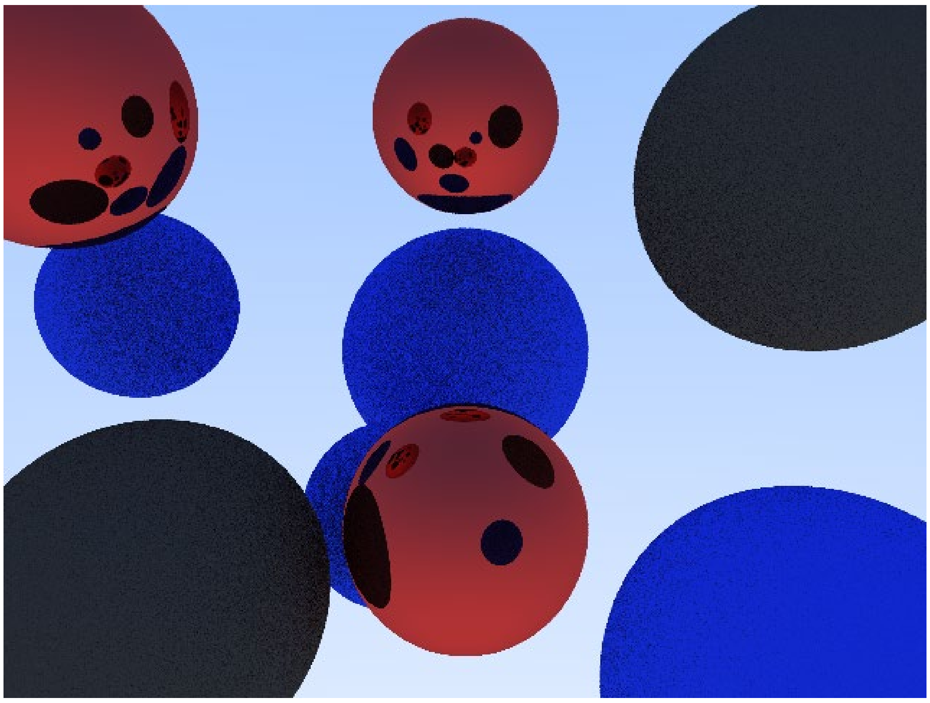


Task Six: object material: specular

The specular material was pretty similar to the diffuse one. Another subclass of “material” called “specular” was created for this. This time the “scatter”-bool has a similar task, but instead of creating a randomly scattered ray, it returns a reflected ray. When there are three specular spheres added, the image looks like this:

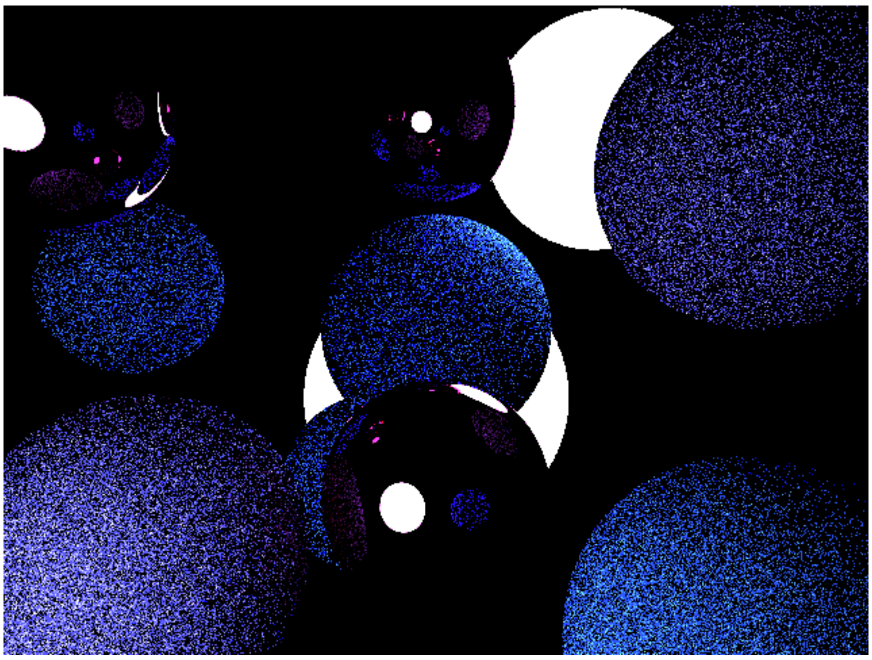


Unfortunately, the recursion limit also did not make a difference here. The example with a limit of 16 yielded the following result (other results are also in the folder):

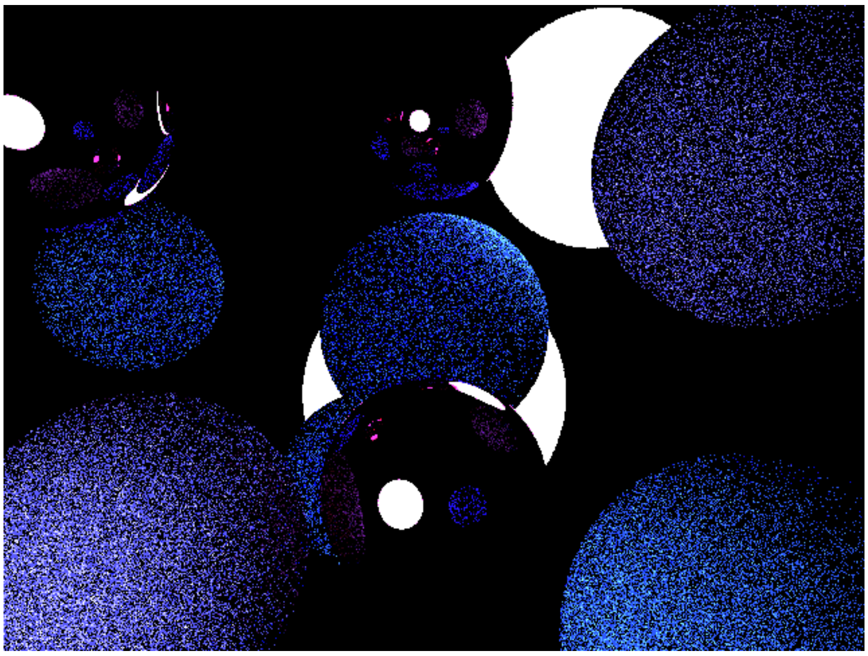


Task 7 (optional): object material: emissive

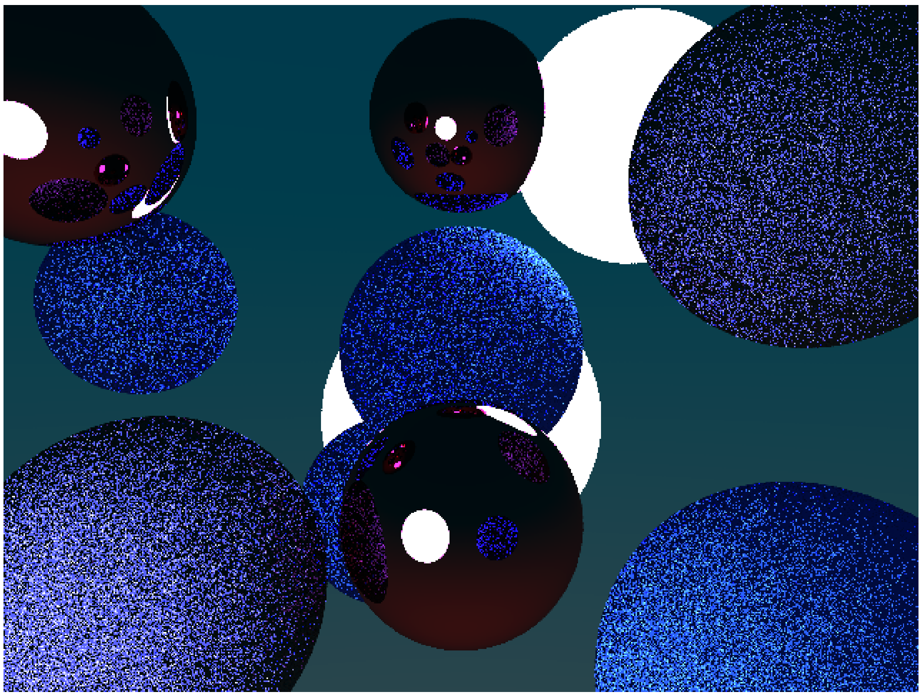
Because the background is the only light source in this image, another material subclass called "emissive" was created. This material introduces another bool called "emits" which just returns false in the case of any other material. The ray_color()-function also has to be updated. Instead of just checking for scattered rays, it now also checks for emitting light. If the material emits light, the ray color is returned by multiplying the light color and light intensity. These values are set in the constructor of the material. The final image with three added emissive spheres looks like this, when the background color is completely gone and a black background is used (the third sphere is only visible in the reflection, because it is positioned slightly behind the camera):



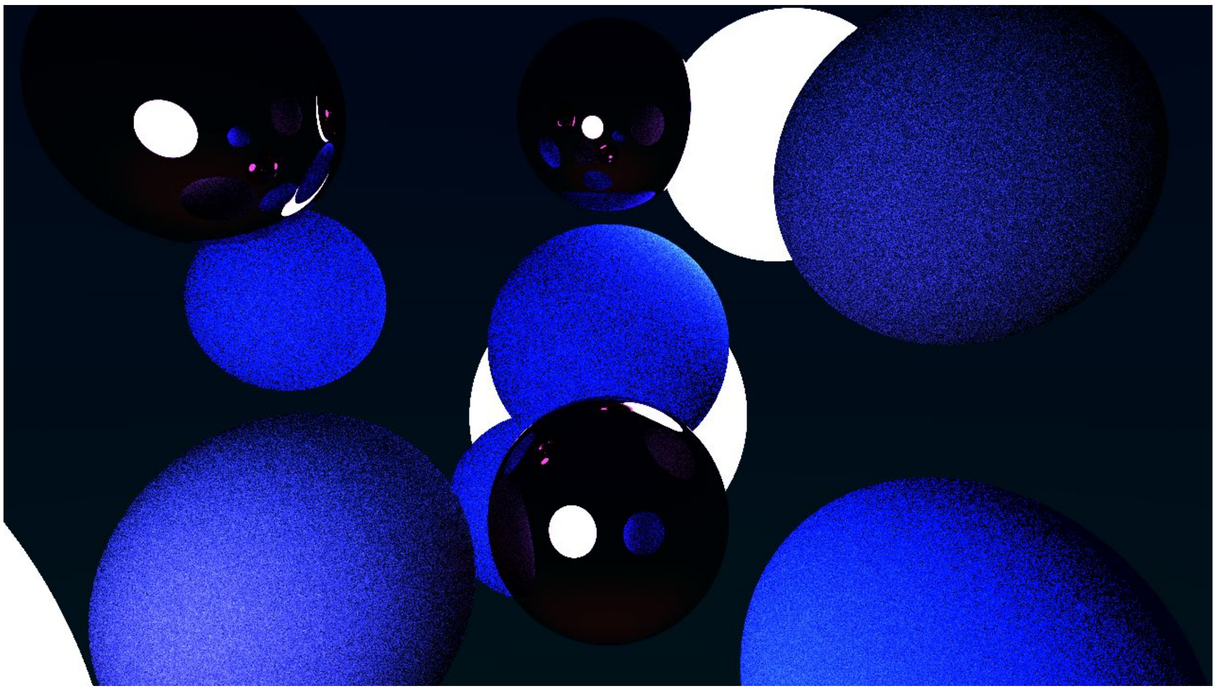
Here, an increase of recursion depth to 16 also did not make a noticeable difference:



When the color gradient background from task two is used, but made slightly darker, the image looks like the following:



I was not really satisfied with this result, because the algorithm took a lot of work and time, so I experimented with different values like the recursion limit or rays per pixel (Anti-Aliasing, variable is called “AAPixel”). Surprisingly, really high values created far better results. So I rendered another image with a higher resolution and different aspect ratio (1280x720, 16:9) and used 48 pixels per ray for my Anti Aliasing. Lastly, I increased the recursion limit to 16000. To make the image a little bit more aesthetically pleasing, I also darkened the color gradient background even further. This led to the following result:



Increasing the recursion limit was probably one of the main factors, why this image looks so much better than the other ones. A limit of 16 was not exactly helping, but huge numbers made the result a lot better.

For inspiration, the link in the help-section of each task was used (https://raytracing.github.io/books/RayTracingInOneWeekend.html)