Raytracer Part 1

In this assignment, you will lay the foundation for a raytracer that you will develop over the next few weeks. **So while this homework can solved with some hacking effort, you have to do it properly, as the next assignments will build on top of this project.**

You will start with some starter code, that implements a very primitive and rather bad raytracer. You will refactor this into an object oriented architecture, to make it ready for further extensions. Finally, you will add some depth perception to the output.

What is a raytracer?

Raytracing is a technique to generate photorealistic pictures on your computer. Let us recap first how regular 3D rendering techniques work, as they are used in video games: in video games your scene is represented with millions of triangles, which are then projected onto a two dimensional surface. This can be realized very efficiently on hardware, because the amount work that has to be done for every polygon is (relatively) simple and can be easily parallelized for a GPU.

Raytracing in contrast represents the scene with geometric objects, like planes or spheres. Then, for every pixel on your screen, a ray is cast through the scene. Then the raytracer calculates with analytical geometry, whether a given ray intersects with any object of the scene. Using an intersection as start point for new rays you can then trace rays to all light sources, to determine the brightness of the object. Or if the object has a reflective surface, you calculate the rays for reflections. You might have heard that modern nVidia GPUs have built-in raytracing support and that there are some games which make use of that.

Over the next few assignments, you will build a raytracer step by step, with every assignment being a checkpoint. It is important to complete all assignments, as the next assignment will build on top of the previous one.

Setup

Download the starter code. This code contains a primitive and ugly raytracer, which renders this scene:

You should first read the code to understand how the raytracer works. You do not need to fully understand how to derive the formula for the intersection. However, you should look up the article mentioned in the source comments regardless. The output format is PGM, because it is very easy to generate.

Running the raytracer

Compile and run the raytracer and look at its output. You should see this:

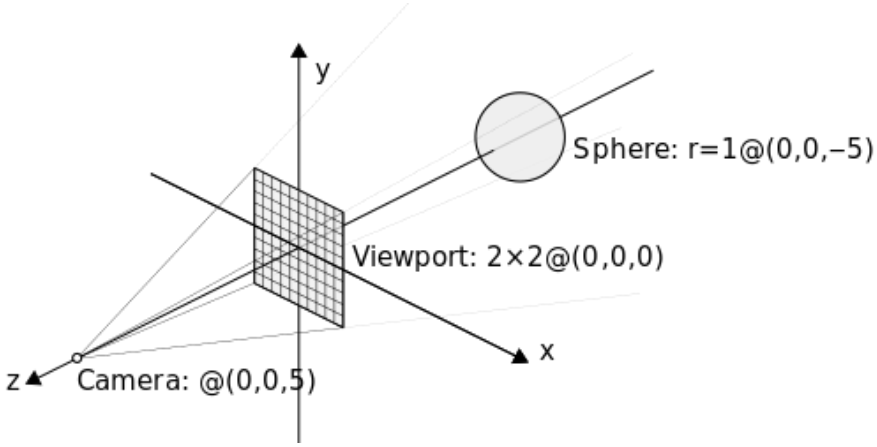


Figure 1: raytracer

```
$ ./a.out
P2
11 11
1
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 0 0 0 0
0 0 0 1 1 1 1 1 0 0 0
0 0 0 1 1 1 1 1 0 0 0
0 0 0 1 1 1 1 1 0 0 0
0 0 0 0 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
```

If we look at the output, we see it starts with `P2`. These must be at the beginning of every PGM file. Then follows the image resolution (11x11) and the maximum brightness (0=black, 1=white). Then the picture follows. You can view them in any decent image viewer (on Windows you can use IrfanView).

Refactoring

We now want to refactor the raytracer into an extensible object hierarchy.

Geometry

You first want to introduce a better representation of points, vectors and rays. Points and vectors are very similar, as they both store three doubles for the

x, y and z coordinates. A ray is a composite of a start point and a vector for the direction. Create structs for point and vector and a class for for rays. The ray is a class, because it has the invariant, that the direction vector is always normalized. Put these classes in a directory `geometry`.

Now replace all `double[3]`-arrays in your main with your pointer and vector classes, and create rays with `camera` as start point and `ray_direction` as direction. You need to define operations between two points to get a vector from the first point to the second, adding vectors together, scaling vectors with a scalar, the dot product between two vectors and functions to calculate the length and normalize vectors.

Scene

As next step, you need to introduce a better encapsulation of spheres. Therefore create a subdirectory `scene` and create a sphere class in there. A sphere has a point that is the center and the radius. Define a function `intersections`, that takes a ray as argument and returns a vector of doubles. These doubles are distances of the intersections measured from the start-point on the ray.

> **Example:** a ray starts at (0;0;-5) and there are two intersections with the sphere at (0;0;4) and (0;0;6), the resulting distances are 9 and 11.

Move the code, that calculates `delta`, `b`, `c` and `discriminant` to the `intersections` function. You can calculate the distances of the intersections like this:

```
distance1 = (-b + sqrt(discriminant)) / 2;
distance2 = (-b - sqrt(discriminant)) / 2;
```

These two doubles are the distances we were looking for. However, we only want those of those two distances, that are positive. If they are negative, the sphere is *behind* the camera, but we are only interested in intersections that lay ahead of us. If only one of the distances is positive, the camera is *inside* the sphere. Note the special case, where the discriminant is zero, then the ray is only a tangent and thus there is only one intersection. Look at the pictures in this article.

Put these two distances into a vector and return it. Refactor the main method to use the sphere class.

Output

The raytracer should not depend on a single output format. Hence, you need to create a class, that saves a two dimensional array as PGM image. Create a directory `image` and a class `PGMOut` with a method `save` with one argument `vector<vector<int>>`. Each inner vector contains a row of brightness values, and the outer vector contains all rows. You can get the dimensions of the picture from the size of the vectors. Print the PGM picture to standard out.

Refactor the main method, to store the pixels in a vector, such that they can be saved as PGM by the `PGMOut` class.

Raytracer

Now all that is left is the raytracer component. Create a new directory `tracer` and add a class for the raytracer itself. The raytracer contains one function `to_raster` with a parameter for resolution and returns the vector for `PGMOut`.

Move the remaining code from main into your `to_raster` function. Parse the resolution from the command line arguments, such that `./a.out 128` generates a 128x128 picture.

Your main should now look like this:

```
int main(int argc, char *argv[]) {
    Raytracer tracer;
    PGMOut out;
    int resolution; // parse from argv[1]
    out.save(tracer.to_raster(resolution));
    return 0;
}
```

Brightness values

If you run the raytracer now, you should get exactly the same output as before:

```
$ ./a.out 11
P2
11 11
1
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 0 0 0 0
0 0 0 1 1 1 1 1 0 0 0
0 0 0 1 1 1 1 1 0 0 0
0 0 0 1 1 1 1 1 0 0 0
0 0 0 0 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
```

The raytracer does not have light sources yet, but we can fake a light source by using the distance values of the intersections as brightness values. The closer the intersection to the camera, the brighter it should be displayed.

Modify your raytracer: the sphere returns a vector with intersection distances. Find the smallest distance in that vector (the one that is closest to the camera)

and calculate the brightness from it as indicated by the following code snippet:

```
double brightness = 10 - distance;
if (brightness < 0) brightness = 0;
int pixel_brightness = static_cast<int>(brightness * 255);
```

This works, because smallest distance between the sphere and the camera is 9. Then we scale this to a color range of 0-255. Now modify `PGMOut` to emit `255` as maximum brightness value. If you run your raytracer now, you should get this output:

```
./a.out 11
P2
11 11
255
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 157 182 157 0 0 0 0
0 0 0 157 222 239 222 157 0 0 0
0 0 0 182 239 255 239 182 0 0 0
0 0 0 157 222 239 222 157 0 0 0
0 0 0 0 157 182 157 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
```

If you run the raytracer at higher resolutions, it will generate this picture:

Submission

Submit your code to gradescope. Running `make` in your makefile directory must yield an executable `a.out <resolution>`, which will generate above picture. Only files named `*.h`, `*.hpp`, `*.cpp`, and `Makefile` are accepted. **If we find that all you have done for this project is modifying `main.cpp` instead of performing the refactoring, we can give you a 0, even if you pass the gradescope submission.**

Project structure

If you followed these instructions, your project should look similar to this:

```
.
├── geometry
│   ├── Point.cpp
│   ├── Point.h
│   ├── Ray.cpp
│   ├── Ray.h
```

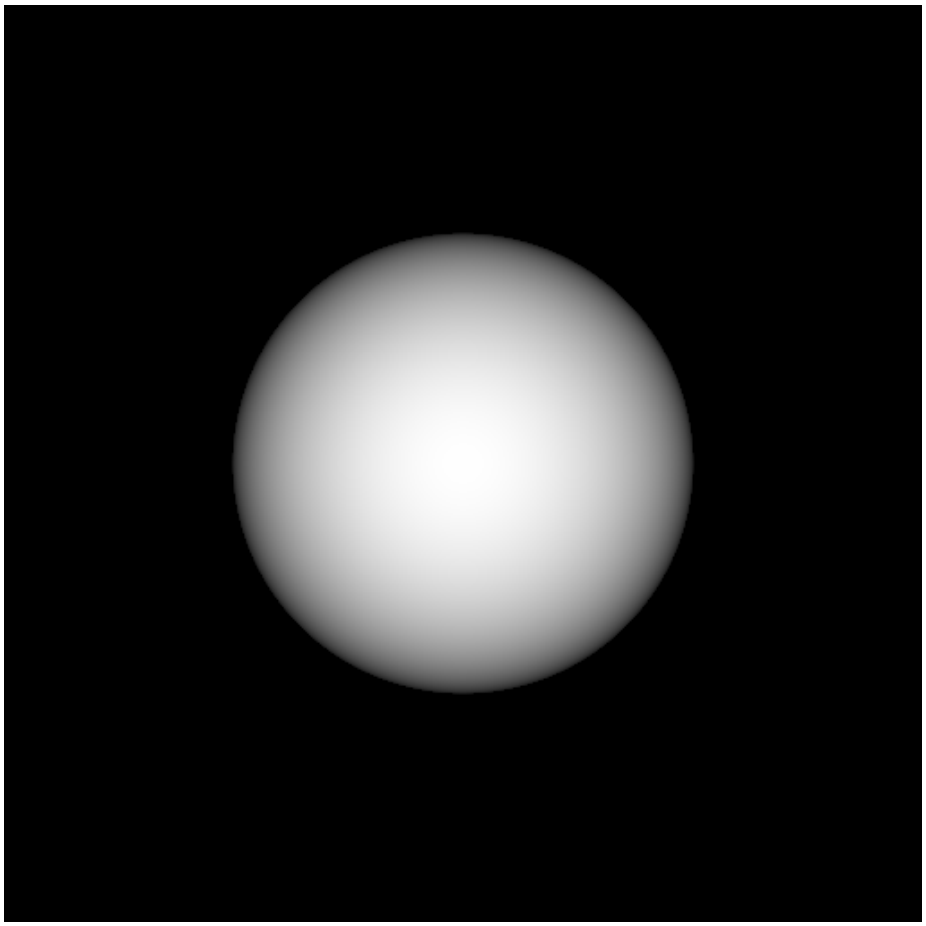
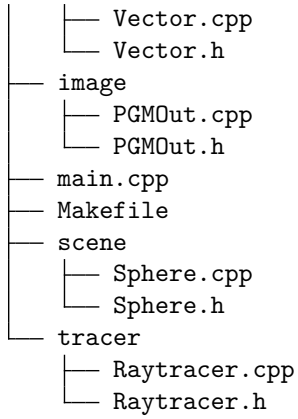


Figure 2: sphere512


Vector.cpp
Vector.h
image
PGMOut.cpp
PGMOut.h
main.cpp
Makefile
scene
Sphere.cpp
Sphere.h
tracer
Raytracer.cpp
Raytracer.h