Ray tracing in Euclidean space

Background

Ray tracing is a method used in computer graphics to simulate photorealism. There are many different versions of this algorithm, but they all focus on tracing rays of light through a scene. We calculate the intersection points of these rays with objects in the scene and then either bounce them in a different direction or refract them through an object if it is transparent. We can repeat this several times, as this also happens in the real world. A ray finally reaches the eyes of a viewer (called a *camera*) and we can color a pixel in the color that the ray has accumulated while traveling through the scene. In general, this algorithm is quite computationally expensive and will not run in real time without heavy optimizations. However, the computational intensity also depends on the choice of parameters, the number of objects, etc.

The goal of this project is to implement a version of ray tracing and render a few simple scenes.

Task desciption

For creating our scene we will use objects that can represent many different shapes, but have a very simple and general mathematical representations: surfaces, given as solutions of the equation

$$f(x,y,z)=0. \tag{1}$$

Examples of such objects are:

(1) Planes given with the equation
$$ax+by+cz=d$$
(2) Spheres given with the equation
$$(x-x_0)^2+(y-y_0)^2+(z-z_0)^2=r^2$$
where (x_0,y_0,z_0) is the center and r is the diameter. We can also get elipsoids and other second order surfaces with similar equations.

(3) Graphs of functions of two variables. Let $z=u(x,y)$ be a function of two variables. We can rewrite the equation as
$$z-u(x,y)=0$$
and we get an equation of shape (1).

Let $T_0(x_0,y_0,z_0)$ be a point and $\vec{v}=(a,b,c)$ a direction. We represent a light ray with equations

$$\begin{aligned} x(t)&=x_0+at,\\ y(t)&=y_0+bt,\\ z(t)&=z_0+ct, \end{aligned} \tag{2}$$

where $t\geq 0$ is a parameter. We start in the point (x_0,y_0,z_0) (where $t=0$) and move along the ray by increasing t by small increments, until we detect the change of sign of the function $f(x,y,z)$ from the equation (1). This means that the ray has hit an object. Here we have to be careful when choosing the increment for t: if it is too big we might miss intersections for some cases and if it is too small the algorithm will run for too long. When we find an intersection on the interval $[t_1,t_2]$ we need to find a better approximation. We use Newton's method to solve the equation

$$g(t):=f(x_0+at,y_0+bt,z_0+ct)=0, \tag{3}$$

with the initial approximation in the middle of the interval: $t = (t_1 + t_2)/2$. The derivative of the function (3), which we need for Newton's method, can be obtained from the formula

$$g'(t) = af_x(x_0+at, y_0+bt, z_0+ct)+bf_y(x_0+at, y_0+bt, z_0+ct)+cf_z(x_0+at, y_0+bt, z_0+ct),$$

where f_x, f_y and f_z are partial derivatives of f with respect to x, y and z, respectively.

When we calculate the intersection of the ray with a surface (let us denote it with $T_1(x_1, y_1, z_1)$) we get the normal vector to the surface at point T_1 as

$$\vec{n} = (f_x(x_1, y_1, z_1), f_y(x_1, y_1, z_1), f_z(x_1, y_1, z_1)).$$

Now we can obtain the direction of the reflected or refraced ray of light with the use of elementary geometry using vectors $\vec{v}$ and $\vec{n}$.

In real world each light source emits practically infinite light rays, which is of course too computationally expensive. To deal with this we can instead trace each ray of light in reverse - starting from the camera. This reduces the number of light rays just to the number of pixels we want to render.

We can represent a light in the scene with a point. Such a light is called a *point light*. You can try out some different scene coloring/lighting options:

(1) Calculate the intersection of a ray with an object and determine the color of the pixel based on the cosine of the angle between the normal and the vector between the intersection and the light source.
(2) Calculate the intersection of a ray with an object and determine the color of the pixel based on the cosine of the angle between the reflected ray and the light source.
(3) Calculate the intersection of a ray with an object and send another ray towards the light source to determine if this point is obstructed by another object (lies in the shade). In this case you can color the point darker.
(4) **Advanced, optional:** bounce off rays from objects in random directions and color them only if they hit the light source (here you have to represent the light as an object, for example a sphere, as a single point is very unlikely to be hit with a random ray). This will create a very noisy image with only one iteration, so you will have to send multiple rays from each pixel and average the results. This method is called *path tracing* and yields more realistic results but is way more computationally expensive than the normal ray tracing described above. You can take a look at the first half of the video [2] to get an idea on how you can implement this type of randomized reflections.

You are free to do practically anything after implementing the basic ray tracing. You can implement multiple ray reflections, allow for refractions to create glass-like object, define different materials of objects, add more advanced lighting, introduce some randomness to the algorithm, make rendering faster with the use of a GPU... You are also encouraged to find different sources to help you with this project. We recommend you take a look at [1] which is a popular website that guides you through the implementation of a ray tracer (it uses a different approach than presented here, but it should be helpful anyways).

References

[1] Shirley, Peter, *Ray tracing in one weekend* Amazon Digital Services LLC, 2018. `https://raytracing.github.io/`
[2] Lague, Sebastian, *Coding adventure: Ray Tracing.* `https://www.youtube.com/watch?v=Qz0KTGYJtUk`