{EPITECH}

RAYTRACER BOOTSTRAP

YOUR CPU GOES BRRR, BUT IT'S THE BOOTSTRAP

RAYTRACER BOOTSTRAP

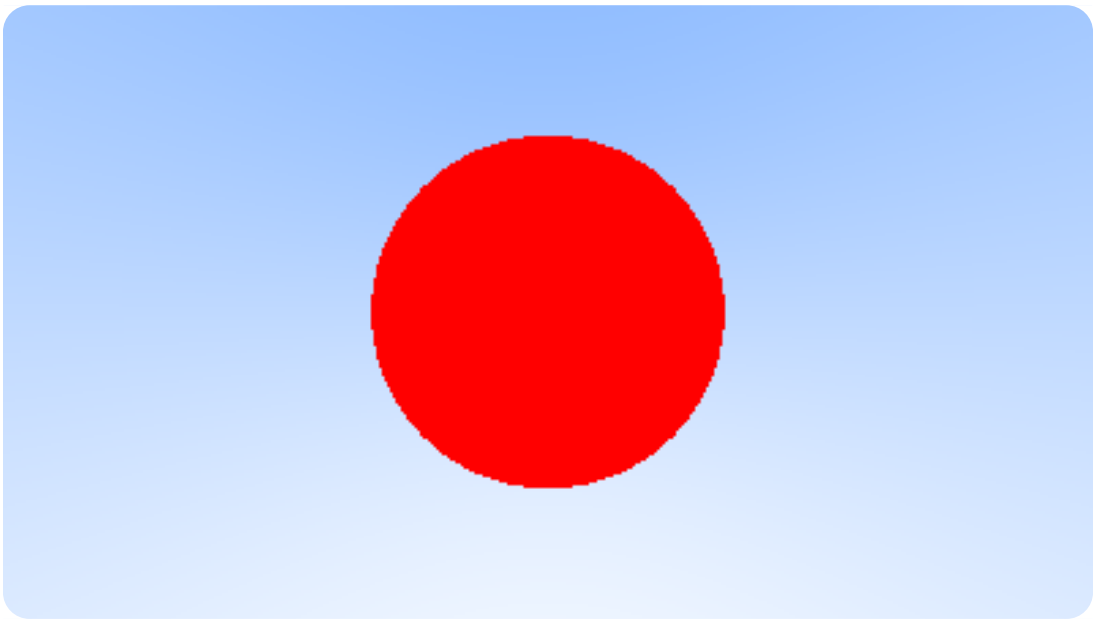
Before doing raytracing you will first have to learn how to do raycasting. And be careful not to confuse any of those with raymarching.

But what is the difference between raytracing and raycasting ?

Raycasting is a process in which intersection points are computed analytically, using mathematical formulae, in order to generate an image.

Raytracing is a more complex rendering technique which tries to reproduce the path light uses and where it hits objects in a scene. Intersection points between rays of light and surfaces may be computed using raycasting. Raytracing also computes secondary rays to collect data used for calculation of reflected or refracted light. It may also use multiple rays per pixels for anti-aliasing.

During this bootstrap you will learn how to compute the intersection between a ray and a sphere to produce an image looking like that:



Step zero - The Vector3D and Point3D classes

The Vector3D

A ray tracer generates digital images by casting rays into a scene. A ray consists of two attributes: its origin and its direction. We need a mathematical object to hold these properties.
In a `Math` namespace, create a `Vector3D` that contains:

- ✓ Three `double` public attributes for the X, Y and Z components
- ✓ A default constructor initializing the components to 0
- ✓ A constructor taking the 3 components as parameters
- ✓ A `length` method that returns the length of the vector

Remember to use `= default` where applicable!

`Math::Vector3D` being a mathematical concept, cleverly overload arithemtic operators to simplify its usage:

- ✓ Add `+`, `+=`, `-`, `-=`, `*`, `*=`, `/` and `/=` operators taking a scond `Vector3D` as parameter, adding/subtracting/muliplying/dividing each componenent respectively
- ✓ Add `*`, `*=`, `/` and `/=` operators taking a `double` as parameter, multiplying/dividing each componenent by the given value

Add a `dot` method to your class, computing the dot product of two vectors.

You could create a templated `Vector<N>` class to handle `N`th dimension vectors.

The Point3D

In the `Math` namespace, create a `Point3D` class that contains:

- ✓ Three `double` public attributes for the X, Y and Z components
- ✓ A default constructor initializing the components to 0
- ✓ A constructor taking the 3 components as parameters

There may only be a few select operations between a `Point3D` and a `Vector3D`. Overload the correct arithmetic operators so that we can add a `Vector` to a `Point` (`point + vector`). The operators must return an instance of `Point3D`.

Step one - The rays

Once your `Vector3D` and `Point3D` classes are ready, create a `Ray` class in a `RayTracer` namespace that contains:

- ✓ A `Point3D` public attribute for its origin
- ✓ A `Vector3D` public attribute for its direction
- ✓ A default constructor with default constructed attributes
- ✓ A constructor taking a `Point3D` and a `Vector3D` as parameter for origin and direction

All these operations could be handled using a generic `Matrix` class. 'member 102architect? … You still can go a long way in this project without the need of matrices, though.

Step two - Creating the sphere

Create a `Sphere` class in the `RayTracer` namespace that contains:

- ✓ A `Point3D` attribute for the center of the sphere
- ✓ A `double` attribute for the radius of the sphere
- ✓ A constructor that will initialize the two attributes of your class
- ✓ A `hits` method that takes a `RayTracer::Ray` in parameter and returns true if the ray passed as parameter intersects with the sphere (see below)

To do the method `hit` you'll need to use a Quadratic Equation. 'member 104intersection? This section breaks down the hows and whys of the mathematical formulae that you will need to use. Pay attention, you will need the reasoning for other primitives.

There are equations that any point in a given geometric object satisfy. We call them parametric equations. Below are the parametric equations of the sphere and a ray:

- ✓ Sphere of radius R with origin `(0, 0, 0)`: $x^2 + y^2 + z^2 = R^2$
- ✓ Ray with origin O (point) and direction D (vector): $P = O + D * k$, with k be an arbitrary number in iR

A point $P(P_x, P_y, P_z)$ is on the sphere of radius R with origin `(0, 0, 0)` if and only if $P_x^2 + P_y^2 + P_z^2 = R^2$. That same point P is on the ray defined by O and D if and only if there exists a k for which $P = O + D * k$.

#hint(A point at the intersection of the sphere and the ray would satisfy both these equations.)

We can rewrite the equation of the ray as follows:

$$\begin{cases} x = Ox + Dx * k \\ y = Oy + Dy * k \\ z = Oz + Dz * k \end{cases} \quad (1)$$

Since a point at the intersection of the sphere and ray satisfies both equations, we can add the equation of the sphere to the mix:

$$\begin{cases} x = Ox + Dx * k \\ y = Oy + Dy * k \\ z = Oz + Dz * k \\ x^2 + y^2 + z^2 = R^2 \end{cases} \quad (2)$$

The goal is to find a k that resolves the equation: $x^2 + y^2 + z^2 - R^2 = 0$. If you write down that equation set and substitute x, y and z, in the end you get an equation of the form:

$$a * k^2 + b * k + c = 0$$

This is a quadratic equation that can be solved by calculating a discriminant d:

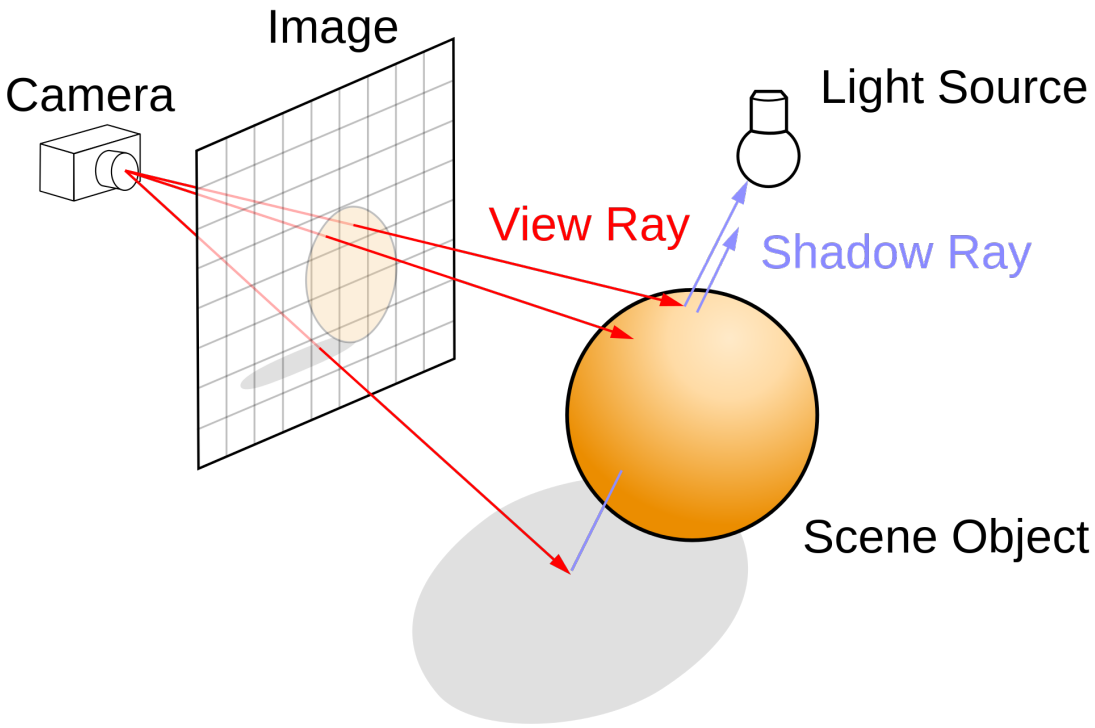
$$d = b^2 - 4 * a * c$$

Step three - The camera and the screen

Now we need a camera in the scene; the camera is the eye. It takes into account multiple parameters: position, rotation, resolution, FOV (field of view), …

By default, the camera will be positioned in `0 = (0, 0, 0)` and will stay the same for each pixel. However the rays will be cast in different directions for each pixels.

In the 3D world, there will also be a screen. The rays will be cast from the camera to each pixel on the screen, one after another. Here is a visual explanation of this process:



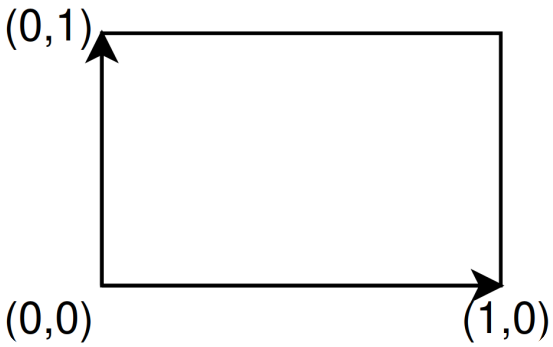
The Rectangle

Create a `Rectangle3D` class that contains:

- ✓ A `Point3D` attribute named `origin` for the bottom-left corner of the rectangle
- ✓ A `Vector3D` attribute named `bottom_side` representing the vector from the bottom-left corner of the rectangle to the bottom-right corner of the rectangle
- ✓ A `Vector3D` attribute named `left_side` representing the vector from the bottom-left corner of the rectangle to the top-left corner of the rectangle

The astute reader may have noticed our class rather represents a parallelogram.

We now need a way of getting the coordinate of an arbitrary point in the rectangle. To help us achieve that, we'll map our rectangle to a space where `(0, 0)` is the origin, `(1, 0)` the bottom-right corner and `(0, 1)` is the top-left corner (see diagram below). In that space, `(0.5, 0.5)` will be the center of our rectangle.
This new 2D coordinate space in the rectangle is different from the 3D space. We will use it to map more clearly our 2D pixels from the image/screen in the 3D space. To each 2D pixel corresponds a (single)? point in the 3D space.



Add a `Point3D Rectangle3D::pointAt(double u, double v)` method with both `u` and `v` in range `[0;1]` that returns the 3D coordinates of the point at the given location in our rectangle.

The Camera

Create a class `Camera` in the `RayTracer` namespace that contains:

- ✓ A `Point3D` attribute named `origin` for the camera's origin point that is initialized to `(0, 0, 0)`
- ✓ A `Rectangle` attribute named `screen` for the screen
- ✓ A `ray` method that takes two doubles `u` and `v` as arguments that represents a point of the screen (using `pointAt`). The method will return a ray, going from the camera to the coordinates `u` and `v` of the image.

Remember to have your class comply with the Coplien form.

The `ray` method will be called repeatedly for each pixel of our screen and will return a ray going from the origin point (the camera) to the current pixel `(u, v)` of the image.

Now we can create a main that look a little something like this:

```
int main()
{
    RayTracer::Camera cam;
    RayTracer::Sphere s(RayTracer::Point3D(0, 0, -1), 0.5);
    for (/* go through the Y axis */) {
        for (/* go through the X axis */) {
            double u = /* get a certain amount of X */;
            double v = /* get a certain amount of Y */;
            RayTracer::Ray r = cam.ray(u, v);
            if (s.hits(r)) {
                // something
            } else {
                // something else
            }
        }
    }
}
```

Step four - Colorful world

Now that we know when a ray collides with a sphere, we can draw, one way or the other, a color on the screen. You are going to focus on creating a .ppm file by writing basic image metadata on the standard output and then write all your pixels on the standard output. Additionally, don't forget to redirect the output in a file when you run the program:

```
Terminal
~/B-OOP-400> ./raytracer > output.ppm
~/B-OOP-400> head output.ppm
P3
400 400
255
0 0 255
0 0 255
0 0 255
0 0 255
0 0 255
0 0 255
0 0 255
```

Write a function `write_color` that takes a `Math::Vector3D` as parameter which represents the color of your pixel and write it on the standard output as shown in the example above.

If a ray hits the sphere, call this function with a red color to print a red pixel, otherwise call it with a blue color. Then, you will get a similar result to the example at the beginning of this bootstrap.

A standalone function, really? You'll surely need to rework that later

Step five - Going further

Keep in mind that everything you did in this bootstrap is **ONE** way of doing raytracing and you may find other methods online. Also, you are vigorously encouraged to rework a few things.

Geometric transformations are also quite useful and you can add them in your class:

- ✓ `translate`: take a `Math::Vector3D` as parameter and move the vector in the 3D space in the direction (and distance) specified by the parameter
- ✓ `rotateX`/`rotateY`/`rotateZ`: take a `double` degree angle as parameter and rotate the vector around the X/Y/Z axis

v 1.1