{EPITECH}

INTERSTONAR

BOOTSTRAP

INTERSTONAR

During this bootstrap and the entirety of the project, it is heavily recommended that you use a graphical library in order to visualize what you compute.

Part I: Simulating celestial bodies: the N-body problem

Let's have fun with physics...

There are 3 laws of motion in Newton physics. These state that:

- ✓ Every object perseveres in its state of rest, or of uniform motion in a right line, unless it is compelled to change that state by forces impressed thereon.
- ✓ The change of motion of an object is proportional to the force impressed; and is made in the direction of the straight line in which the force is impressed.
- ✓ To every action, there is always opposed an equal reaction; or, the mutual actions of two bodies upon each other are always equal, and directed to contrary parts

Moreover, Newton's law of universal gravitation describes the gravitational force by which every object is attracted by any other object:

$$F = G\frac{m_1 m_2}{r^2}$$

This formula is great, but we can rewrite it using vectors. For two bodies named 1 and 2, we have:

$$\vec{F}_{1/2} = -\vec{F}_{2/1} = G\frac{m_1 m_2}{\left|\vec{r}_{1/2}\right|^2}\hat{r} = G\frac{m_1 m_2}{\left|\vec{r}_{1/2}\right|^3}\vec{r}_{1/2}$$

with: $\vec{r}_{1/2} = \vec{r}_2 - \vec{r}_1$ and $\hat{r} = \frac{\vec{r}_{1/2}}{|\vec{r}_{1/2}|}$

While the original equation gave us a **scalar**, this equation gives us a **vector**. This will be more appropriate for our use case.

Using these laws and equations, we can try to predict how the position of objects in a scene is going to evolve over time:

- ✓ In a scene with only one object, this is trivial: the first law of motion states that it will either have a uniform motion in a straight line, or will stay put.
- ✓ With two objects, we have equations that simply reduces the problem as a pair of unique bodies. Therefore, we can predict exactly how the system is going to evolve over time.
- ✓ From three bodies and more, we don't have any exact solutions for the general cases, and the evolution of the system will entirely depend upon the initial conditions (we say that the system is chaotic). This is known as the three-body problem or, more generally, as the N-body problem. Therefore, we will need to make estimations using **numerical analysis**.

Computation method

With all these equations in mind, we can try to compute the acceleration that any object of our scene is going to have at every moment in time. This requires to sum the force exerted by every other object on it.

Using the second law of motion, we know that, for every object i:

$$\vec{F_i} = m_i\vec{a}_i = \sum_{i,j=1}^{n} G\frac{m_i m_j}{|\vec{r}_{i/j}|^3}\vec{r}_{i/j}$$

with $\vec{a}$ the acceleration of the object

We can deduct that the acceleration exerted on the body is:

$$\vec{a}_i = \frac{\vec{F_i}}{m_i}$$

We are now almost ready to compute everything, we just need a method to update every important value of our system. We shall use Euler's integration method in order to approximate our positions and speed:

$$\vec{r}_{i+1} = \vec{r}_i + \vec{v}_i dt$$

$$\vec{v}_{i+1} = \vec{v}_i + \vec{a}_i dt$$

Better integrators exist such as Runge-Kutta 4 (rk4 for the initiated), which can be implemented as a bonus!

Going further

If you've been educated to algorithmic optimization, you shall notice that this algorithm has a algorithmic complexity in $O(n^2)$, which can be problematic, especially when trying to simulate a large amount of objects.

Using Newton's third law of motion, we can decrease the amount of computations done, although keeping a $O(n^2)$ complexity.

$$\vec{F}_{1/2} = -\vec{F}_{2/1}$$

However, there is an ever better way that reduces the complexity to $O(n \log n)$, known as the **Barnes-Hut simulation**

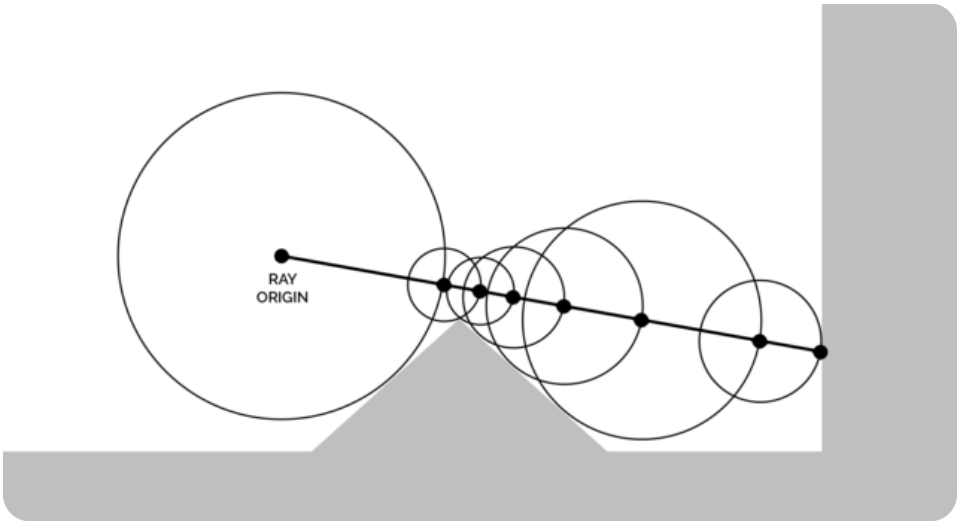
Part II: Ray Marching

Let's have fun with maths...

Ray marching is a class of graphical rendering methods that uses light rays to check if an object is seen in the scene or not, just like *ray casting* or *ray tracing*.
However, while these methods use *parametric equations* that can be more or less complex to solve, ray marching uses a different approach.

There are different ways to make ray marching, here we will focus on **Sphere-assisted ray marching**.
In order to check if a (light) ray intersects any shape, we will start at the initial position of the said ray, and try to find the closest intersection point between this point and all the shapes, in any direction (thus drawing a sphere, just like its name suggests).
Once this point is found, if the ray of the sphere we just made is smaller than some threshold,

we consider there is a hit, otherwise we move the position of the ray on its direction vector. Then, we repeat this process until either finding an intersection with an object, or until we reach a maximum number of iterations.



This method relies on computing the closest point on the surface of any shape to the position of the ray. This is achieved by computing a Signed Distance Function (or SDF).

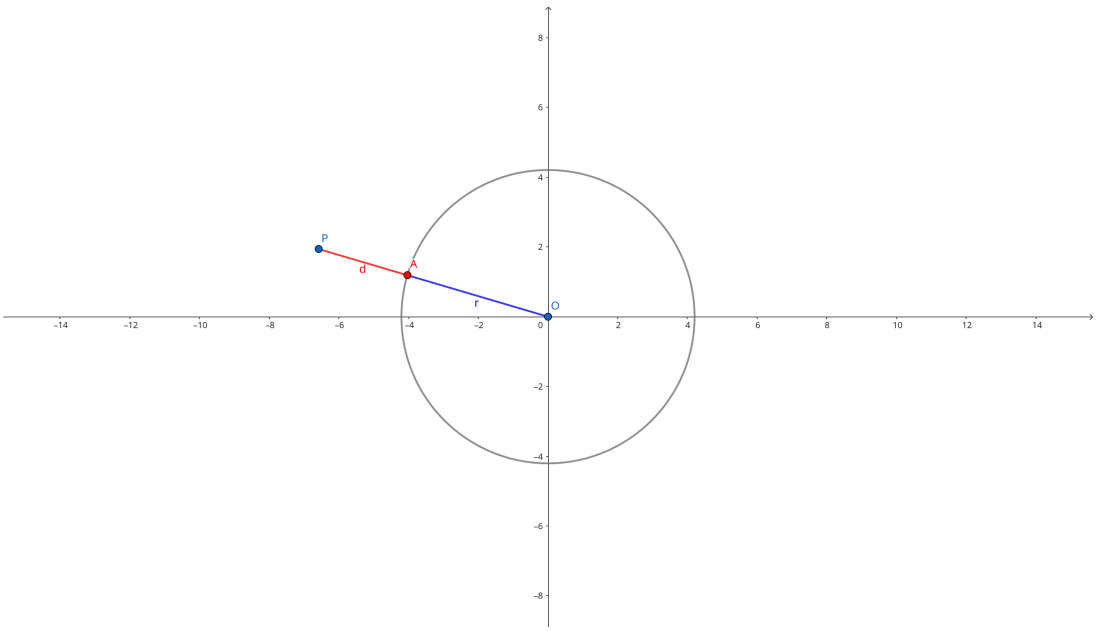
In this bootstrap, we will try to find a way to compute some of these SDF using the power of *maths*.

Sphere SDF

If we have a point P whose coordinates are represented by the vector $\vec{P} = (P_x, P_y, P_z)$, a circle centered at the origin O of radius r, and d being the distance between P and A, the closest point on the Sphere, then:

$$\left|\vec{P}\right| = \sqrt{P_x^2 + P_y^2 + P_z^2} = d + r$$

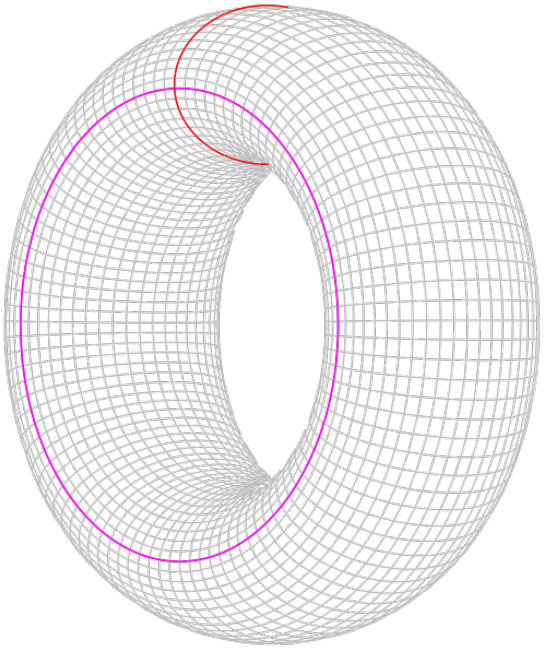
$$d = \sqrt{P_x^2 + P_y^2 + P_z^2} - r$$



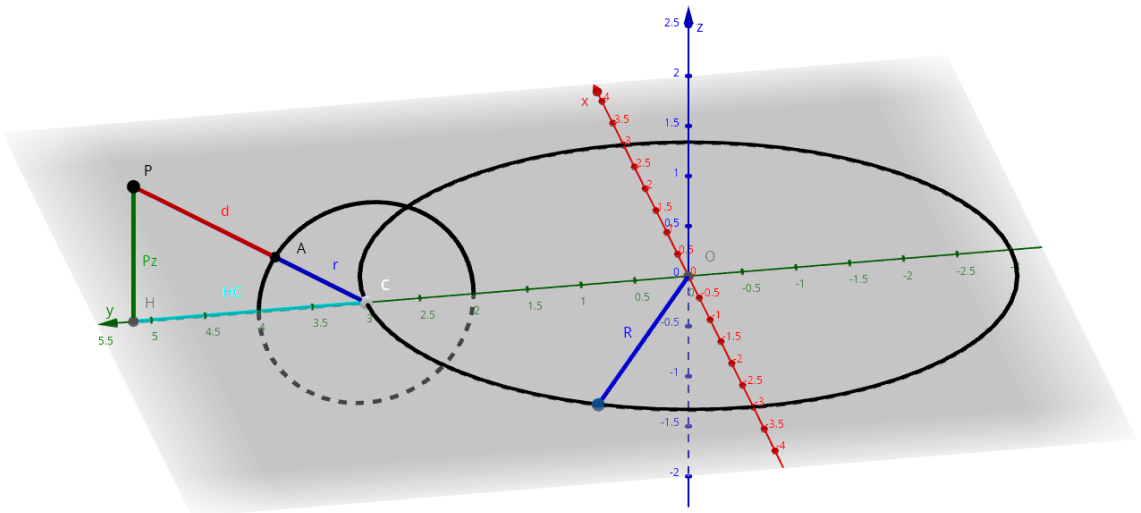
Could you generalize it for any sphere not centered on O ?

Torus SDF

Now that we can compute the SDF for a circle, let's take a look at more complex shapes, starting by the torus.
A torus can be described as a sphere circle of radius r, whose center follows a line forming a bigger circle of radius R.



Again, let's assume a torus centered on the origin O. The shortest distance d between any point P of space whose coordinates are represented by the vector $\vec{P} = (P_x, P_y, P_z)$ can be obtained easily using the Pythagorean theorem.



Using these values, the Pythagorean theorem and the figure above, we can see that:

$$HC = \left|\vec{H}\right| - R$$

$$d = \sqrt{P_z^2 + HC^2} - r$$

Again, try to generalize this equation for a torus not centered on O

Cylinder SDF

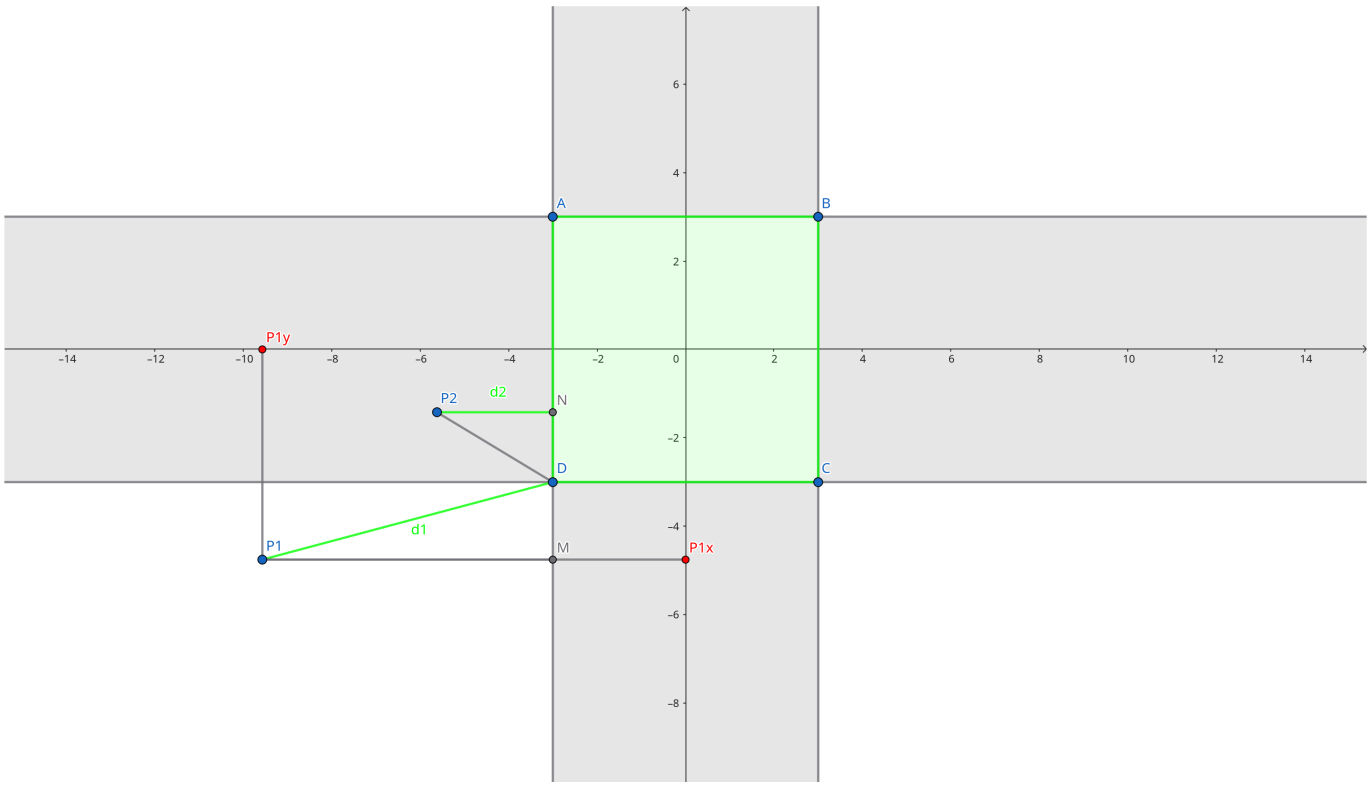
Now that we are used to the technique for evaluating an SDF on trivial shapes, the cylinder SDF should be a piece of cake… If we assume an infinite right circular cylinder of radius r around the z-axis.
In that case, one might easily find that the SDF shall be: $d = \sqrt{\left|\vec{P}\right|^2 + P_z^2} - r$

Your mission, should you choose to accept it, is now to tweak this equation for cylinders revolving around any arbitrary axis and, if you can, with limited length.

Cube

Last but not least, the cube. For this one, there are two different cases to handle that are represented in the next figure: the one when the point is in the white area, and the one when the point is inside de grey area.
There are two main ways to handle this: either by checking manually in which case we are, and applying the right formula, or by finding one single formula to handle both cases.

Now that you are a master in Euclidean geometry and that the Pythagorean theorem has no more secrets for you, can you find the SDF function associated with the cube?

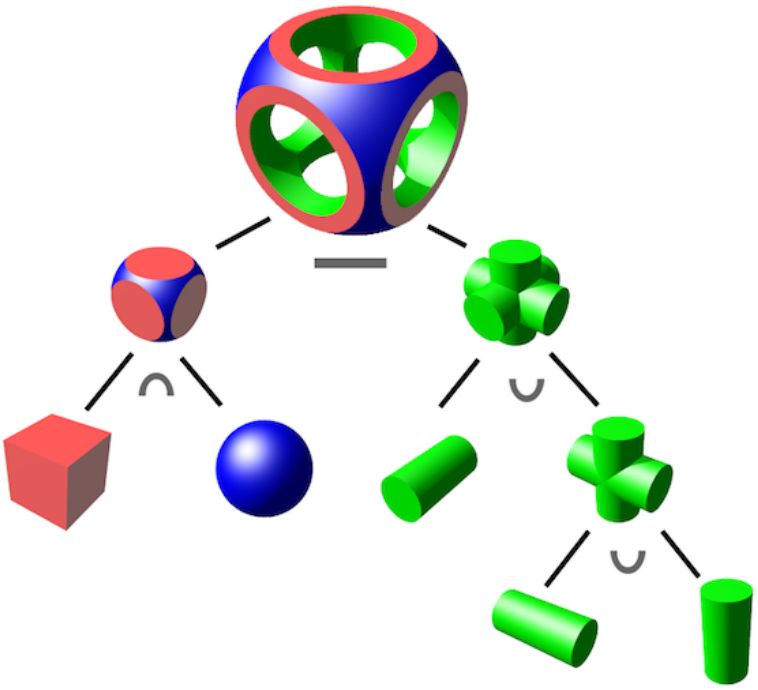


Going further

Now that we have basic SDF functions for our objects, there are still more things to do in order to make our project more interesting:

- ✓ We need to generalize the formulas in the case where the shapes aren't nicely centered in O.
- ✓ It would be nice to handle objects rotation, for example having a torus whose big circle is not parallel to the Oxy plane.
- ✓ You can also add more shapes (like cylinders, planes, prisms, Platonic solids, etc.)

Finally, you should be aware that there is more in ray marching than adding more SDF for new shapes. You can easily apply mathematical operations on multiple shapes (like unions, substractions, intersections, etc) in order to create new ones!



v1