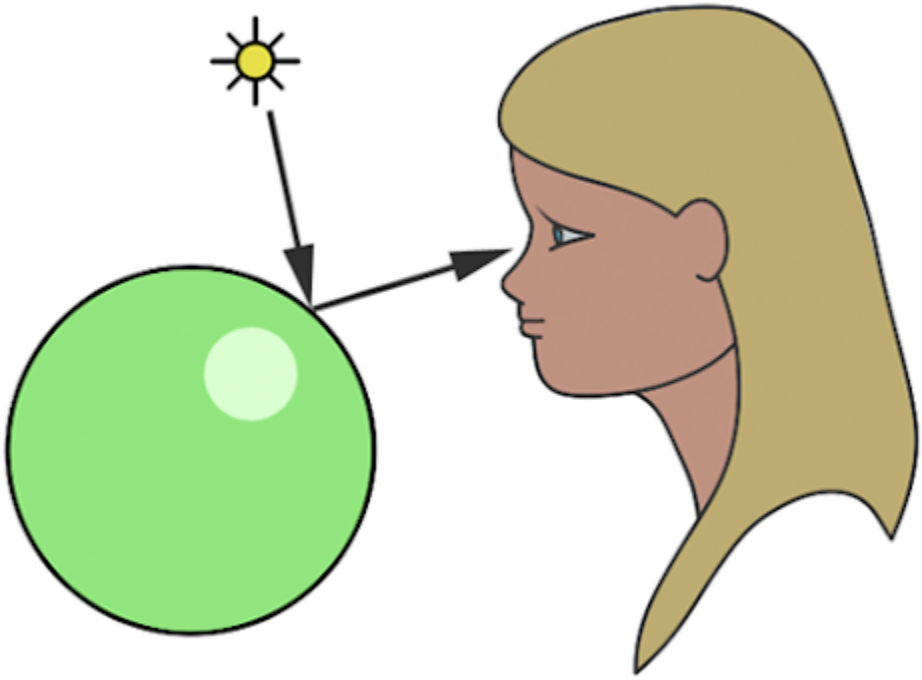
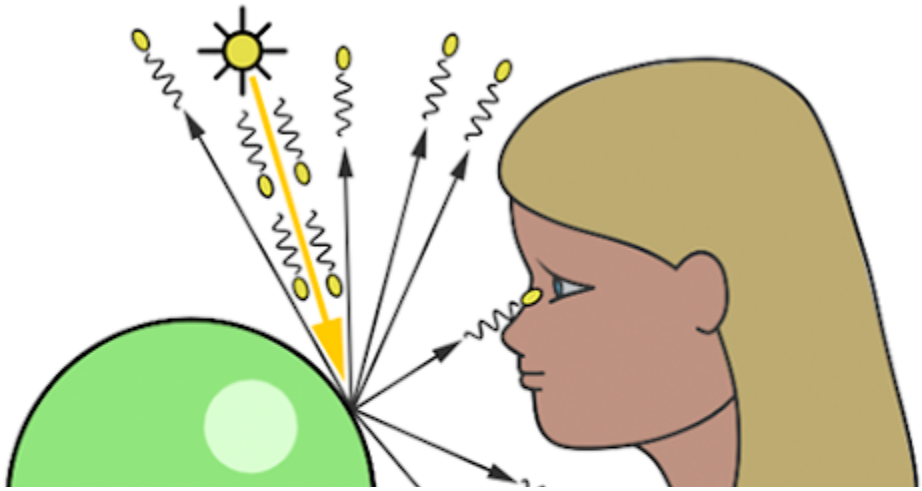
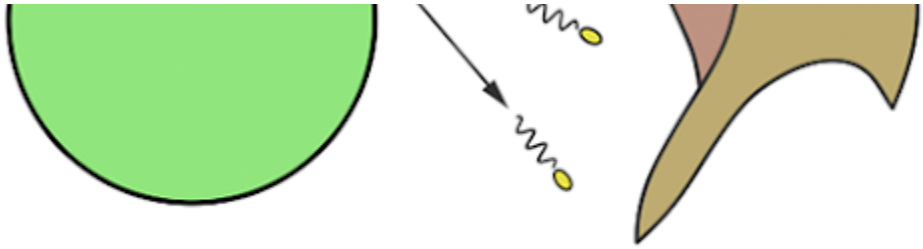
Rayos 3D

Queremos replicar el comportamiento de la luz cuando llega a nuestros ojos luego de impactar en algún objeto.

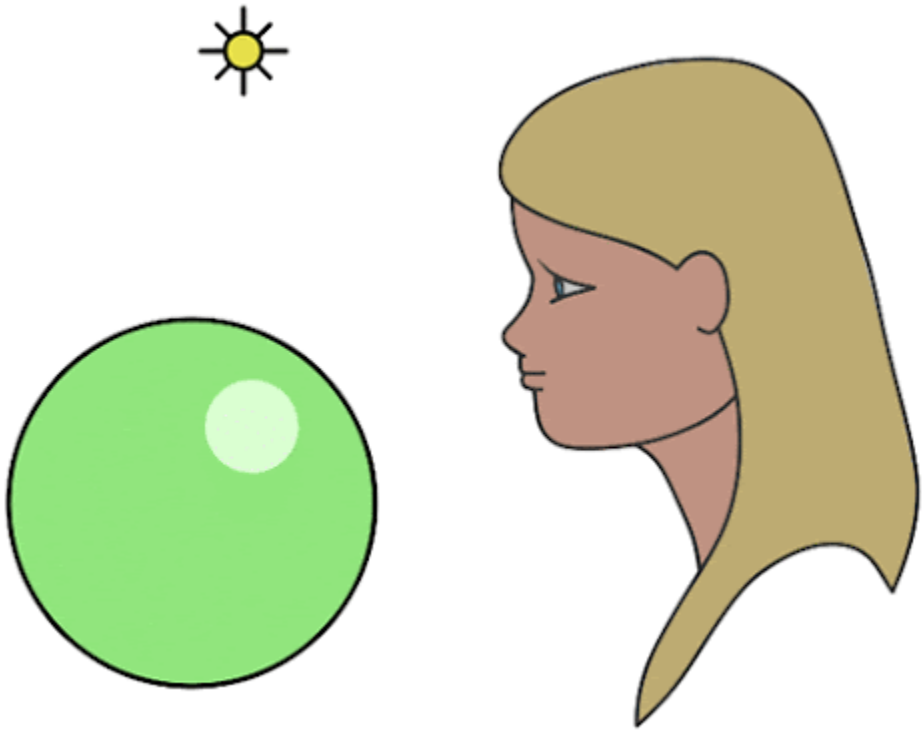


Sin embargo, rápido podemos darnos cuenta de que este proceso es extremadamente complejo, sino imposible, pues no solo son una cantidad infinita de rayos los que salen del origen, sino que una cantidad igualmente infinita de rayos se emiten luego de impactar contra un objeto. Ninguna computadora será jamás capaz de simular un proceso tan complejo.

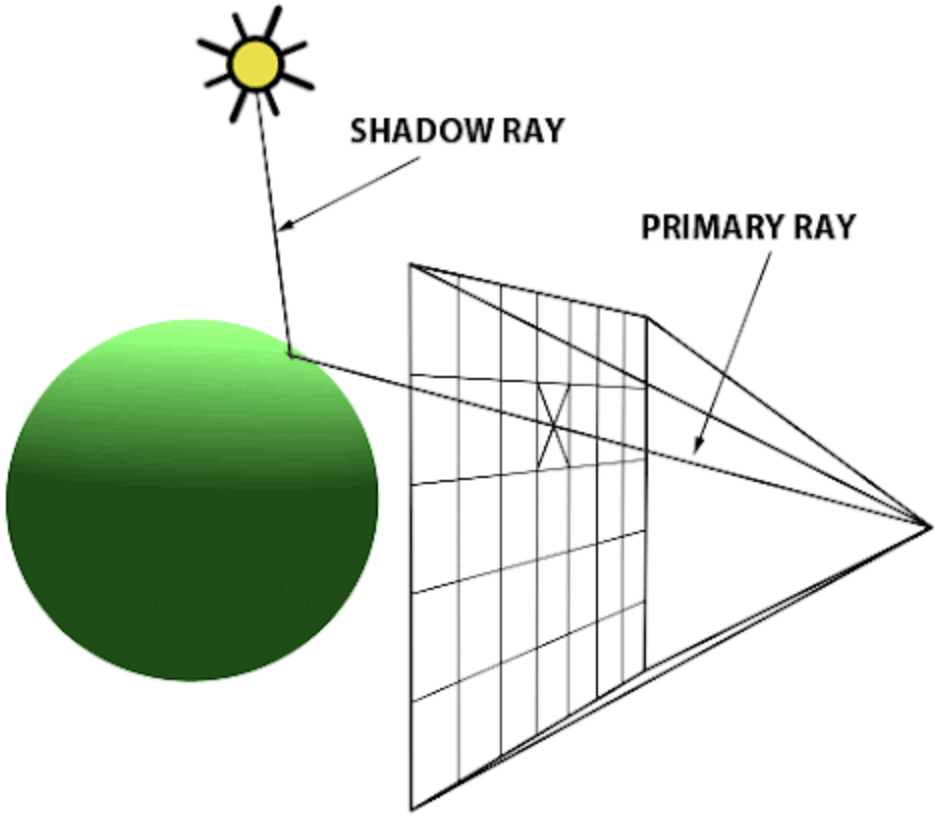




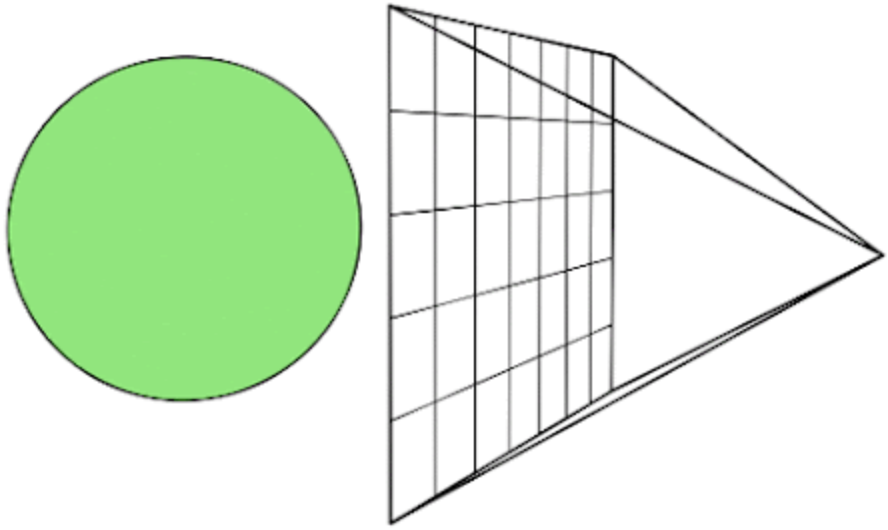
Entonces, ¿qué podemos hacer para simplificar el proceso? Basta con darnos cuenta de que no son todos los rayos los que nos interesan, sino realmente solo los que finalmente llegan a nuestros ojos. La genial idea de un Raytracer es la de invertir el proceso. En lugar de simular rayos que se originan en la fuente de luz, vamos a simular un rayo que sale desde nuestro punto de vista, que impacta con varios objetos y que finalmente llega a la fuente de luz.



Para nuestros fines, basta entonces el lanzar un rayo por cada pixel de nuestra pantalla que va a viajar por el espacio en una pequeña “aventura” y que nos va a traer un color de regreso.



La combinación de todos estos rayos es la que va a describir la forma que tienen estos objetos en 2D. Esta es una proyección de muy alta definición hecha sin hacer ninguna transformación.

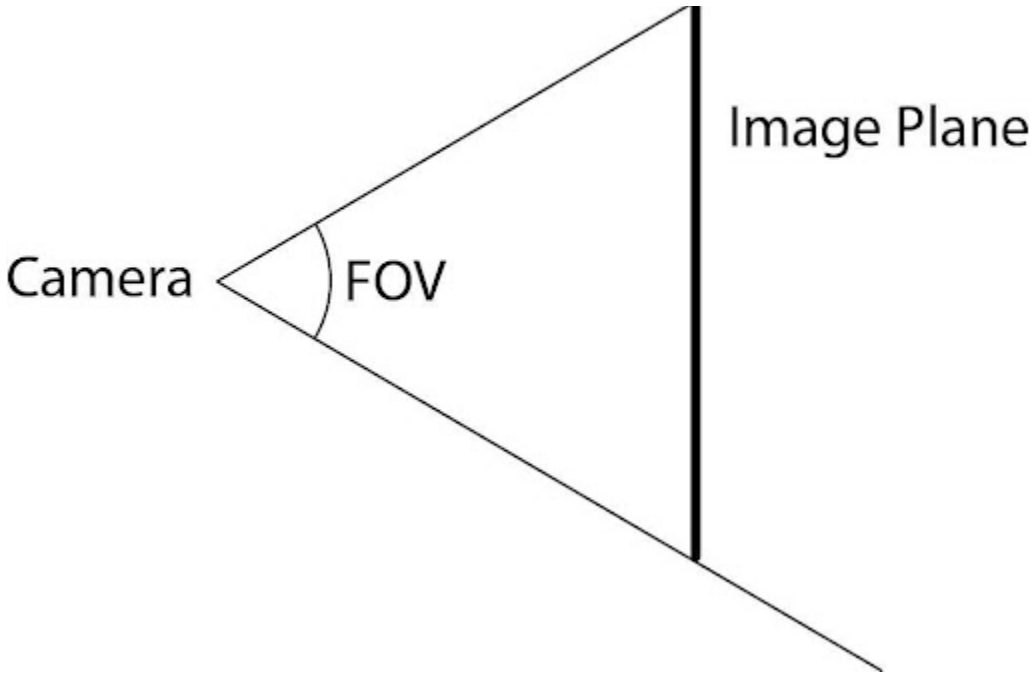


Es decir que para nosotros, un raytracer no es más que un for de dos dimensiones que recorre la pantalla.

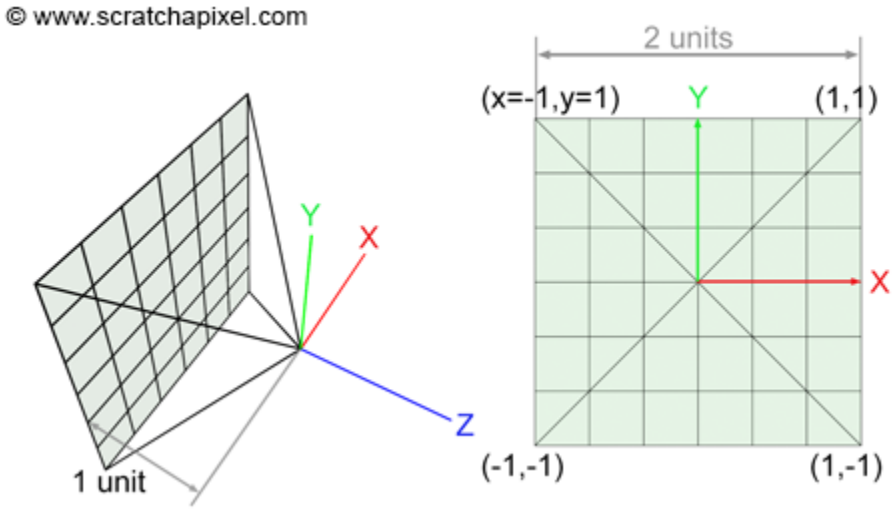
```
pub fn render(framebuffer: &mut Framebuffer, objects: &[Object]) {
    let width = framebuffer.width as f32;
    let height = framebuffer.height as f32;
    let aspect_ratio = width / height;

    for y in 0..framebuffer.height {
        for x in 0..framebuffer.width {
            // Map the pixel coordinate to screen space [-1, 1]
            let screen_x = (2.0 * x as f32) / width - 1.0;
            let screen_y = -(2.0 * y as f32) / height + 1.0;

            // Adjust for aspect ratio
            let screen_x = screen_x * aspect_ratio;

            // Calculate the direction of the ray for this pixel
            let ray_direction = normalize(&Vec3::new(screen_x, screen_y, -1.0));

            // Cast the ray and get the pixel color
            let pixel_color = cast_ray(&Vec3::new(0.0, 0.0, 0.0), &ray_direction, ob
jects);

            // Draw the pixel on screen with the returned color
            framebuffer.set_current_color(pixel_color);
            framebuffer.point(x, y);
        }
    }
}
```

¿Qué quiere decir normalize en este código y por qué es necesario?

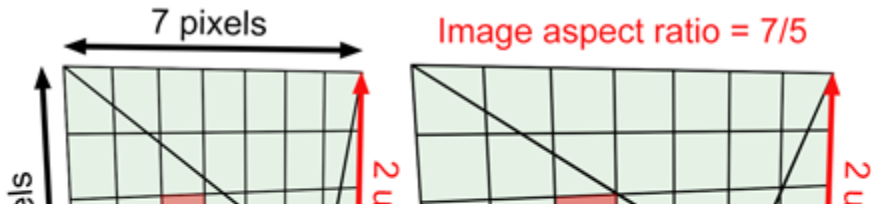
Pueden notar en el código que para poder lanzar el rayo tuvimos que hacer un cambio de coordenas. La manera más simple de comprender el porqué es si recuerdan que nuestros rayos en realidad salen de un único punto que de hecho está detras de la “pantalla”.

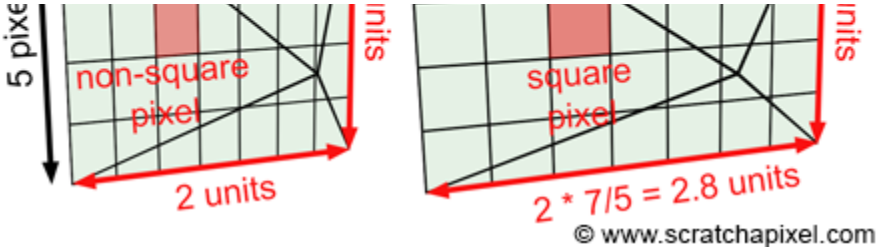


Imaginen que este punto corresponde a (0, 0) para nuestro mundo, y todos los demás píxeles en la pantalla forman un cuadrado que va de [-1, -1] a [1, 1]. Dediquen un poco de tiempo a entender porque multiplicamos el valor de cada coordenada x, y por 2 y luego le restamos 1 para poder lograr esto.



Finalmente, debemos hacer un ajuste relacionado al **aspect ratio** (https://en.wikipedia.org/wiki/Aspect_ratio_(image)) de nuestra pantalla. Esto lo necesitamos hacer porque nuestras ventanas no siempre son perfectamente cuadradas, sino que tienden a ser más horizontales que verticales (extra: ¿por qué son nuestras ventanas más largas horizontalmente que verticalmente?)




5 pixels
non-square pixel
2 units
square pixel
2 * 7/5 = 2.8 units
© www.scratchapixel.com