POPL Milestone #1

Project Name: Ray Tracer in Rust (vs C++)

Group #32

Tanay Doshi	2021A7PS2984G
Aaryan Chauhan	2021A7PS2595G
Smit Bagadia	2021A7PS2406G
Aditya Shah	2021A7PS2454G

Ray-Tracer in Rust:

Problem Statement
Implementing a simple ray tracer in Rust vs the C++ variant and understanding the differences in coding, usability and security for these variants, as a project for the course Principles of Programming Languages.

Inspired from this book:
https://raytracing.github.io/v3/books/RayTracingInOneWeekend.html
Credits to Peter Shirley, Steve Hollasch and Trevor David Black.

Learning objectives: Rust syntax, modules, cargo, crates, references and borrowing, copy, clone, etc. crates, generics and traits, etc.

We need to at least learn how to create images, so we do so in the ppm format. Using the basic colors red, blue and green, we can create patterns and combinations using the double loop (256x256 pixels).

Vec3 struct: We will use this to implement various concepts needed in image generation, from a point in 3D space to a specific color using the combinations of red, blue and green. **We only used aliases for doing this, so we won't get any warnings.**

Notice that vec3 is a small struct, so it is quite efficient to pass it by copy whenever we need to transfer it. Here is an interesting link:
https://www.forrestthewoods.com/blog/should-small-rust-structs-be-passed-by-copy-or-by-borrow/
Key takeaways:

- The code for passing structs by copying is much cleaner than by borrowing. Borrowing temporary values every time for all small structs is impractical.

- Performance isn't affected for small structs whether you pass by copy or by reference.

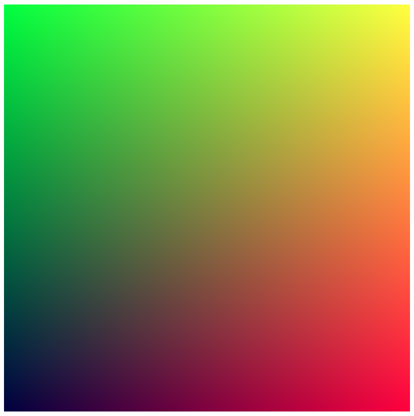
- By default, regardless of passing by copy or by reference, Rust was doubly faster than C++ for this job.

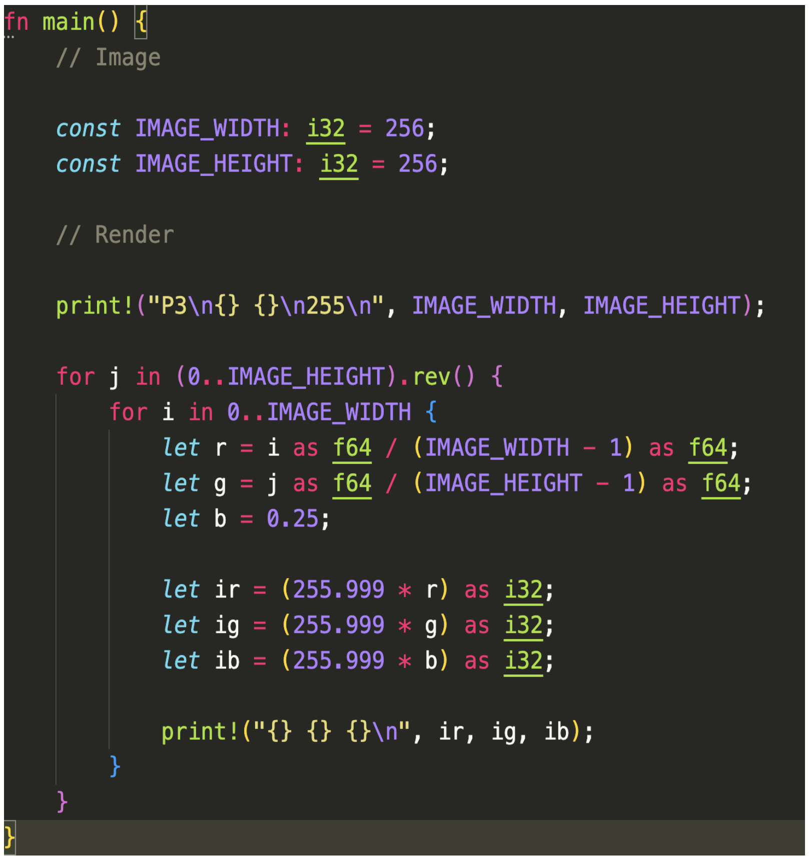
- "Rust is remarkably more efficient at utilizing my CPU's floating point vectors. The difference is enormous! Rust f32 is almost 3.5x more efficient than C++ float. And, for some reason, C++ double is twice as efficient as float!"

This benchmarking reinforces our idea that implementing the ray tracer in Rust is going to be faster than C++ for large renders.

Another important difference to note is that **package handling** is much easier to use and well-implemented than C++. Utilizing any new cargo package requires just one command line instruction. The project is also much better maintained in this way - the Cargo.toml file holds all external dependencies used in the project as opposed to a header file in C++. We have used the rand crate in our project.

1) Creating Our first PPM Image using RUST:

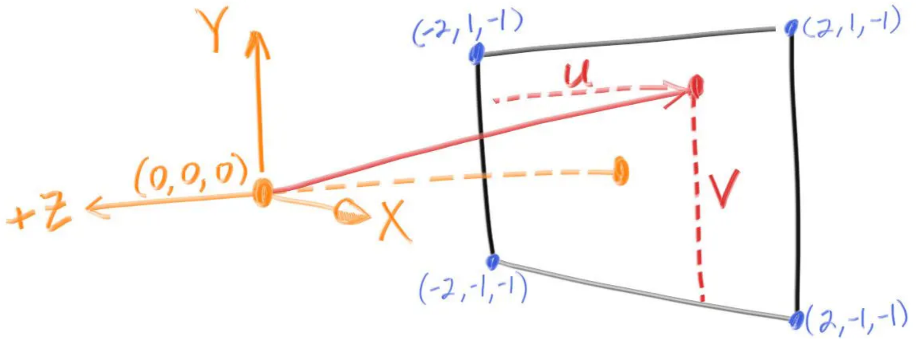




```rust
fn main() {
    // Image

    const IMAGE_WIDTH: i32 = 256;
    const IMAGE_HEIGHT: i32 = 256;

    // Render

    print!("P3\n{} {}\n255\n", IMAGE_WIDTH, IMAGE_HEIGHT);

    for j in (0..IMAGE_HEIGHT).rev() {
        for i in 0..IMAGE_WIDTH {
            let r = i as f64 / (IMAGE_WIDTH - 1) as f64;
            let g = j as f64 / (IMAGE_HEIGHT - 1) as f64;
            let b = 0.25;

            let ir = (255.999 * r) as i32;
            let ig = (255.999 * g) as i32;
            let ib = (255.999 * b) as i32;

            print!("{} {} {}\n", ir, ig, ib);
        }
    }
}
```

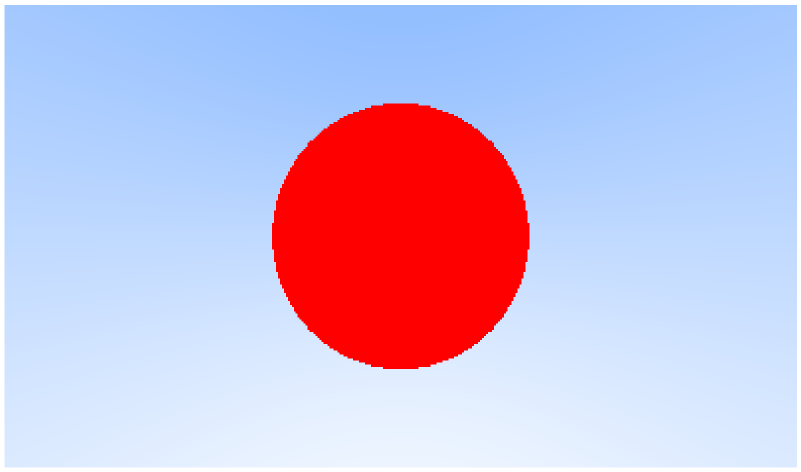
- P3 means colors are in ASCII.
- Code calculates normalized values r, g, and b representing the red, green, and blue components of the pixel.
- The code prints the RGB values of each pixel to the standard output.Each pixel is represented in a separate line.

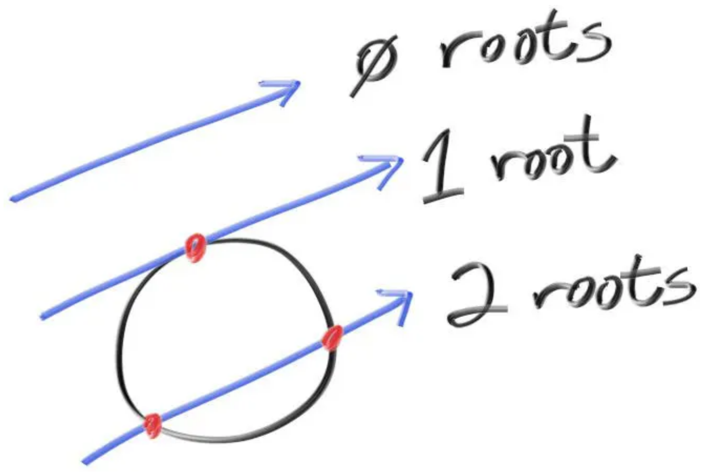
2) Rays and Background:



- Ray tracer sends rays through pixels and computes the color seen in the direction of those rays.
- The involved steps are:
  (1) calculate the ray from the eye to the pixel,
  (2) determine which objects the ray intersects, and
  (3) compute a color for that intersection point.
- The code implements a right-handed coordinate system with the camera at the origin, generating rays for pixels from the upper left corner. It utilizes offset vectors along the screen sides without normalization, ensuring simplicity and efficiency in ray tracing computations.
- The ray_color() function linearly blends white and blue depending on the height of the *y* coordinate.We're looking at the y height after normalizing the vector, you'll notice a horizontal gradient to the color in addition to the vertical gradient.
- BlendedValue=$(1-t)\cdot$startValue$+t\cdot$endValue.

3) Adding a sphere:



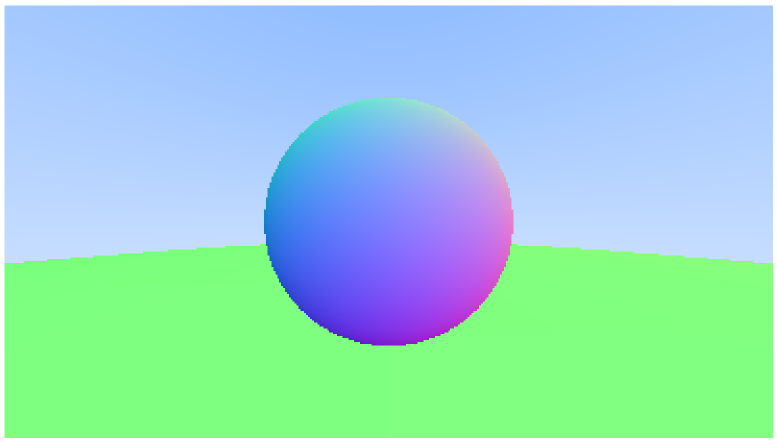
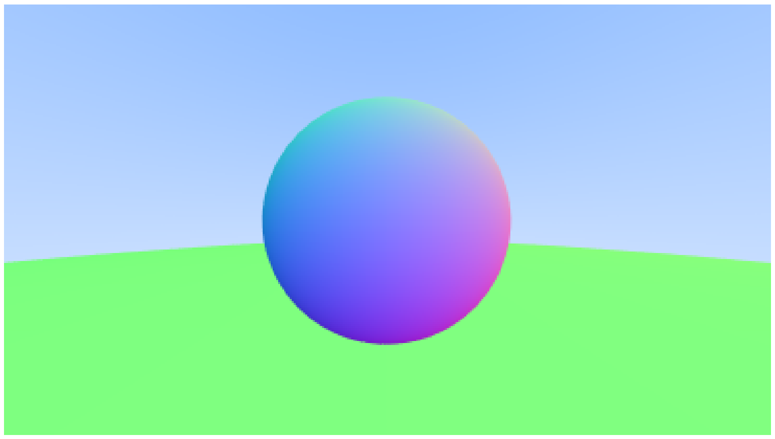
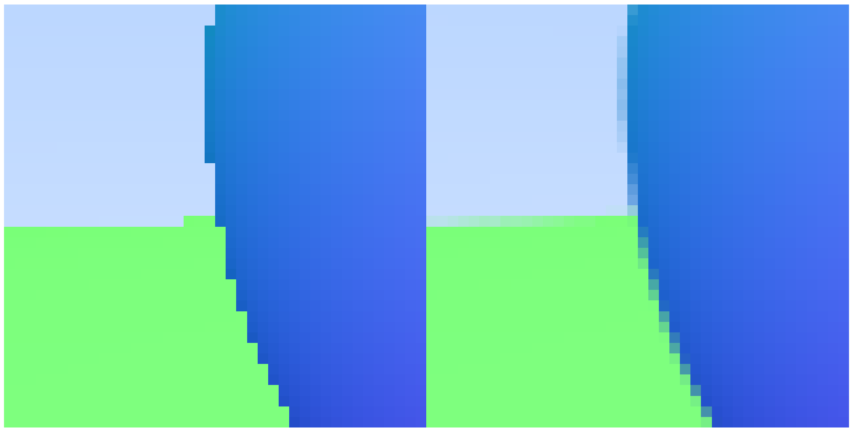


Ray-Sphere Intersection (calculate whether a ray hits a sphere):

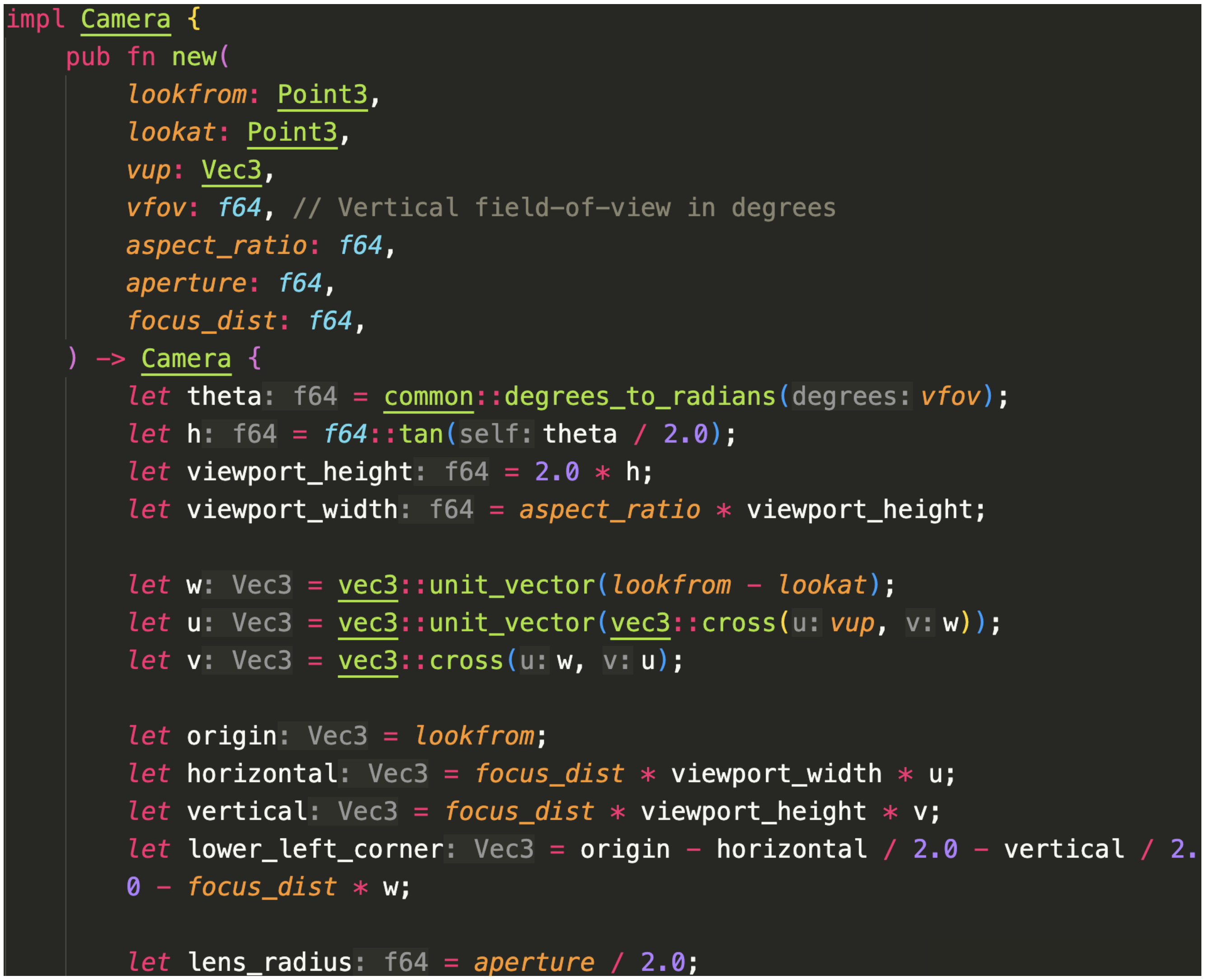
- Equation of sphere centered at origin is $x^2 + y^2 + z^2 = R^2$
- If the given point (x,y,z) is inside the sphere, then $x^2 + y^2 + z^2 < R^2$, and $x^2 + y^2 + z^2 > R^2$ when outside.
- If the discriminant of the ray and sphere intersection is positive then 2 roots (secant), zero then 1 root (tangent) and negative then 0 roots (no intersection).
- We can test this concept by coloring red any pixel that hits a small sphere we place at -1 on the z-axis.

➔ We will soon add things like shading, reflection rays and having more than one object in the render.

4) Antialiasing:

- Antialiasing explains how to create smooth edges in images by blending foreground and background colors. This is done by averaging samples within pixels, ensuring a better appearance. The camera class is designed for future improvements in creating advanced cameras.
- We have used a random number generator which generates a number between 0 and 1.
- We will generate each pixel with multiple samples to give it that effect. We send rays through each of these samples and the colors of those rays are averaged.
- This is slightly heavy for computation (and hence is sometimes not used for very large renders like video game worlds) but some computation is eased by adding all samples then dividing at once in the end.



```
impl Camera {
    pub fn new(
        lookfrom: Point3,
        lookat: Point3,
        vup: Vec3,
        vfov: f64, // Vertical field-of-view in degrees
        aspect_ratio: f64,
        aperture: f64,
        focus_dist: f64,
    ) -> Camera {
        let theta: f64 = common::degrees_to_radians(degrees: vfov);
        let h: f64 = f64::tan(self: theta / 2.0);
        let viewport_height: f64 = 2.0 * h;
        let viewport_width: f64 = aspect_ratio * viewport_height;

        let w: Vec3 = vec3::unit_vector(lookfrom - lookat);
        let u: Vec3 = vec3::unit_vector(vec3::cross(u: vup, v: w));
        let v: Vec3 = vec3::cross(u: w, v: u);

        let origin: Vec3 = lookfrom;
        let horizontal: Vec3 = focus_dist * viewport_width * u;
        let vertical: Vec3 = focus_dist * viewport_height * v;
        let lower_left_corner: Vec3 = origin - horizontal / 2.0 - vertical / 2.
        0 - focus_dist * w;

        let lens_radius: f64 = aperture / 2.0;
```

——------------------——------------——---------Thank You——--------------——--------------——----------