Parallel CPU Raytracer

By Joseph Crown, Caleb Harris, Glenroy Little, and Bautista Solanet

GitHub Repo: https://github.com/Joey-Crown/parallel-raytracing-rs

Introduction

In computer graphics, ray tracing is one of the oldest and most reliable methods for rendering 3D scenes. Countless tutorials exist online which seek to introduce new programmers to the concepts and algorithms necessary to carry out the rendering of static 3D scenes. However many of these tutorials produce a project that will run in a single thread, which, given the advanced architecture of modern CPUs, will limit the performance of what can be an easily multithreaded program. While it is more popular to use GPUs to speed up rendering, our team felt that it would be illustrative and educational to try and optimize a ray tracer to run on a CPU, in order to find the areas where performance would most benefit from running on many threads.

A Raytracer works by casting rays from the camera into the environment. The program checks if these rays hit any objects, from here, rays are cast from the moment of collision to determine if anything is reflected. Rays are also cast to any light sources from the collision to determine how the light affects it. This ends up involving a lot of computations for the computer, especially when we apply it to higher resolutions. This makes raytracing perfect for multithreading.

Strategy

Beyond the opportunity to understand the practical applications of multithreading, our team thought this would be a great opportunity to learn how more modern programming languages implement multithreading. For that reason, we chose to write our ray tracer in Rust, partly because it differentiated us from one of the most popular ray tracing tutorials online. The “Ray Tracing in One Weekend” series of e-books is the first major resource one will come across when learning about ray tracing, but since it is written in C++ we wanted to take on the challenge of converting this to Rust code.

The other benefit of using Rust is that its language implementation of concurrency makes use of one of Rust’s most significant features, memory safety. Rust’s borrow-checker approach to memory management means that shared resources can only be accessed in a thread safe

manner, which reduces the possibility of deadlocks and race conditions. This means that our ray tracer program can be free of most of the popular pitfalls when it comes to multithreading.

Rust may be perceived as a slow language for implementing a raytracer due to its emphasis on safety and memory management, which can lead to overhead in certain cases. Rust's borrow checker and strict ownership model do ensure memory safety and prevent other issues like data races and point reference bugs. However these safety features come with performance costs, especially when it comes to raytracing. Raytracing often relies on frequent memory allocation in large data structures, which will require a fine-grained workaround in lieu of Rust's strict ownership model. Nonetheless with careful code design and optimization, Rust is still a viable choice for building a raytracer, offering the benefits of safety and reliability alongside good performance.

To kick off this project, our first step will involve building a functional ray-tracer in Rust. This ray-tracer will possess the capability to render various objects, primarily spheres, each with distinct properties like size, material, and reflective qualities within a 3D environment. The scene will utilize both positional and directional cameras, along with features such as anti-aliasing and defocus blurring. Once the foundation of the ray-tracer is established, we will explore avenues for implementing parallelism.

In the context of our locally developed ray-tracer, we'll employ several strategies to maximize our parallel processing efficiency. Initially, we'll create a pool of worker threads at the beginning of the rendering process, significantly minimizing the overhead of thread creation and deletion. Additionally, we will seek to dissect the rendering process into smaller more manageable subtasks, such as rendering individual image tiles or processing subsets of rays. Monitoring the runtime after these adjustments will help pinpoint computationally intensive points in the code, which will likely revolve around ray-object intersection tests and the shading calculations. In these critical areas, taking loop-level parallelism or data parallelism as an approach will likely lead to significant growths in speed.

Goals

Our goal for this project is to first create a simple ray tracer in rust, and then to optimize that software using parallelism concepts. If done successfully this should induce a measurably significant deduction in runtime. Our end product will be the old raytracer with no multithreading,

alongside a newer optimized version that has multithreading. Our end deliverable could then include quantifiable data of the time differences between programs on the same scene, which would be a sure measure of completion.

An additional nice-to-have after we reach the minimal requirements for this goal is to potentially upgrade both our ray tracers' capabilities. Currently our scene consists only of a sphere and a generic background, but this current design can definitely be improved in a number of ways–some of the easier improvements could involve adding a variety of scenes and a few extra object types. This goal does come with the potential of adding a substantial amount of work, and the environment may require some expertise with computer graphics, so some aspects of this goal may be outside of our scope.

Tasks

There are 3 main tasks that this project comes down to:

1. Understand rust environment, functionality, and syntax.
2. Create a ray tracer using rust as the primary language
3. Optimize the ray tracer using multithreading as a separate program

Step 2 is straightforward in regard to development. The functionality present in our base ray tracer will be modeled after what is in the "Ray Tracing in One Weekend" series of e-books, and will be capable of running any number of spheres on a simple scene. Step 3 is a generalization of processes and can be decomposed into the somewhat ordered processes:

1. Add thread pooling
2. Parallelize the rendering process
    a. Group by image tiles
    b. Group by subsets of rays
3. Parallelize computationally intensive algorithms
    a. Ray-object intersection tests
    b. Shading calculations
    c. Shadow ray tracing
4. Load balancing

Challenges

Our team was not familiar with Rust when we began this project. However this project is the perfect introduction to the language as it forced us to really consider how to make sure our program was memory safe and thread safe. The Rust documentation refers to the language's approach to multithreading "Fearless Concurrency" because the language provides the tools and methods necessary to tackle multithreading without typical concerns. It was interesting to learn the nuances of how Rust handles shared resources between threads with atomic operations, or conditional variables. Overall the decision to choose Rust was both educational and practical for this project.

The next largest problem lies in the optimization steps. Multithreaded ray tracers usually use very specific and intentional memory management which runs ancillary to rust's memory safe design. Having to work within the bounds of rust's strict ownership model to mimic the process of memory allocation and deallocation will be a large hurdle, and the main hurdle we will likely have to deal with.

Results

Here we will compare a few different methods of parallelizing our main render loop. We will compare these results against a single threaded approach, but also against each other to see what will provide the fastest implementation.