Ray-Tracing with an Automatically Partitioned Bounded Volume Hierarchy.

Brennan Seymour
Math 424 Final Project
bseymour@iastate.edu

Abstract

For this paper, I wrote a raytracer in Rust. I sped it up through two primary optimizations: paralellization and the constructed of a Bounded Volume Hierarchy (BVH) for faster intersections with large numbers of objects.

Introduction

A ray-tracer is a form of image rendering program. It converts a scene of geometric objects into a flat image. True to its name, a ray-tracer generates a multitude of rays, simulating the movement of light particles, and computes the intersections between each ray and each object in the scene. After reflecting off a few objects and into a light source, we can determine the color of a corresponding pixel based off the physical properties of the objects which the ray intersected. Do this like a bazillion times and you can create a really pleasant image.

However, ray-tracing is a very performance intensive rendering method. A massive performance bottleneck arrives as the number of objects in a scene increases, as every ray must check for intersections with every object. Since checking for intersection with a simple bounded volume is very cheap, we can generate a hierarchy of these volumes, (a BVH) and reduce the domain of relevant objects to only the relevant bounded volume. Do this recursively, and the technique can decimate the number of expensive calculations needed to resolve each ray.

Unfortunately, ray-tracing doesn't have very much practical application, besides being really cool. It's only just beginning to be adopted as a serious rendering technique for realtime applications, and that requires software and hardware optimizations far beyond what I'm capable of. However, ray-tracing is really cool. This is my primary motivation in picking this project – It's cool.

I'll be writing the project in Rust, a language that I've recently learned to use. Rust is quickly becoming my favorite programming language, and I'm pretty excited to use it for a serious project. I have confirmed that Rust can be executed on Nova – I just need to compile a statically linked binary, which is very feasible.

Method

As for the process of basic raytracing, I think I covered that fairly well in the introduction. If you'd like to know more about my methods, I invite you to examine the raytracer.rs file on the Github repository. Here I will mostly detail my methods of optimization.

Parallelization

First, parallelization is really straightforward. In fact, raytracing is an embarassingly parallel problem, because it has a massive domain where every calculation can be done entirely independently. So, I

paralellized by splitting the image into horizontal bands and rendering each individually, then merging those bands into a final image before encoding into a PNG file. My main method of paralellism was a Rust library called Rayon. Rayon gives you access to its parallel iterator via a trait. Using this parallel iterator, you can queue up tasks to be distributed to a thread pool. The iterator will then block until its tasks are complete, then return an iterator over your finished elements. This is a great example of fork-join paralellism, and the thread pool allows me to easily overcome dynamic difficulty (some bands take longer than other) by splitting the image into more bands than we have threads.

I also implemented an alternate style of parallelism using Rust's standard library: mpsc. This stands for Multiple Producer Single Consumer, and it is a library which creates channels for cross-thread communication. As the name implies, a sender can be cloned whereas a reciever cannot. So, I split off a number of threads, giving them each a band to render and a clone of our sender to report back through. Then I can join the threads and receive all the finished bands from the reciever held by the primary thread. This mimicks the behavior of MPI, though it communicates between OS threads rather than processes, defeating some of the scaling properties of MPI. This impementation does not address dynamic difficulty.

Bounded Volume Hierarchies

The other, more interesting optimization that I pursued was a Bounded Volume Hierarchy. This is an alternate way of collecting objects in a scene that allows for far faster lookups while testing for intersection. Typically, a scene consisting of many primitives would be represented by a simple list of those primitives. Then, to intersect with it, we test intersection with every primitive in the list and select the nearest intersection. To alleviate this, we might partition the set of elements into smaller groups of primitive, which are contained by an axis-aligned bounding box (AABB). Then, we can test if a ray intersects with this AABB very cheaply. If the ray intsersects, we can test all the primitives in this box. More importantly, if the ray doesn't intersect with our AABB, we can safely discard it and all its primitives.

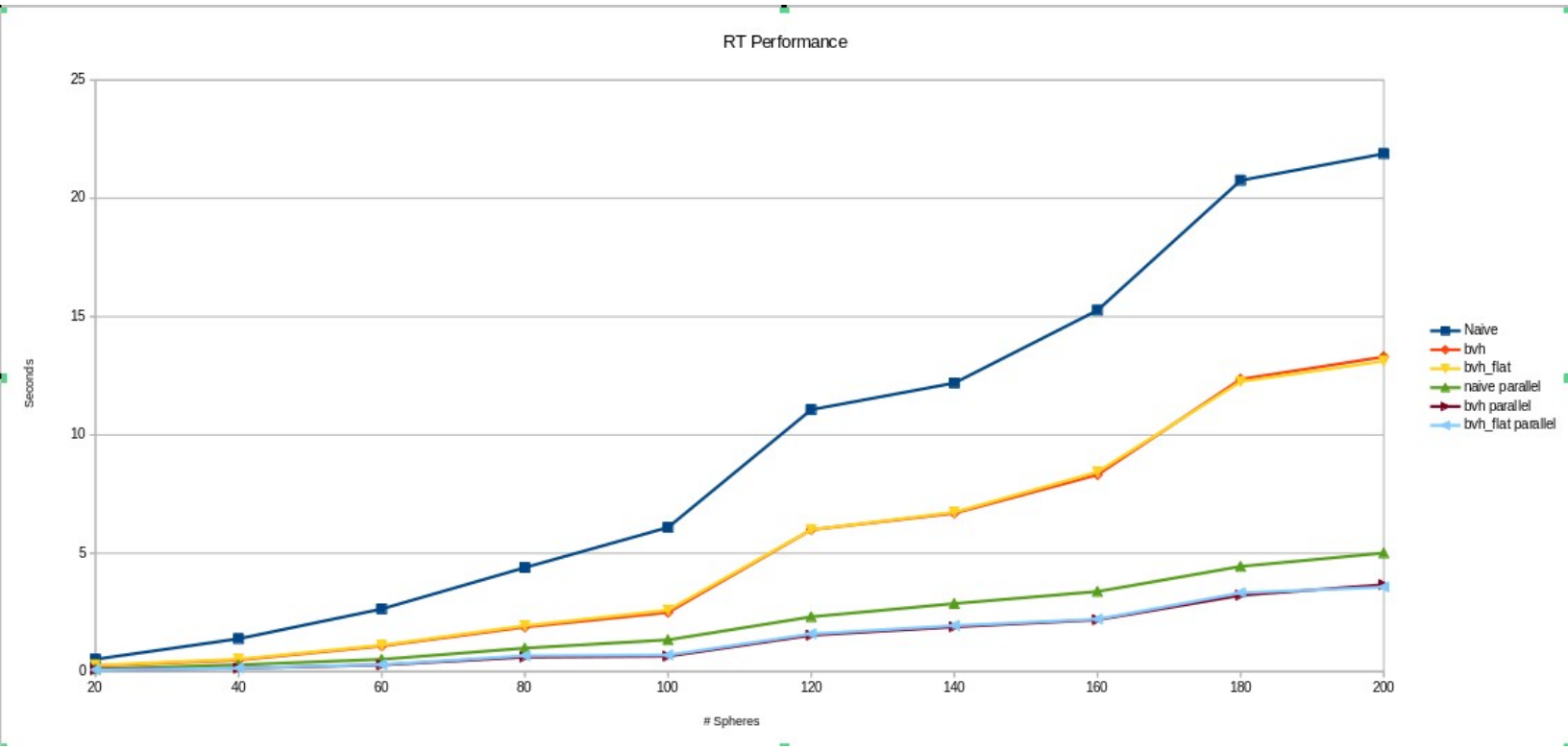
My method takes a collection of primitives and partitions them in two. First it determines minimal AABBs for each primitive along with that box's centroid. Then a minimal AABB is computed to contain all the primitives' centroids. A plane is constructed to split this box along its longest axis, across the box's centroid. Then all the primitives are split into two collections, for each side of our partition. Each half then computes an AABB which is the union of all its primitives AABBs. The end result is that we get a node with a splitting plane, and two collections which are contained in their respective AABBs. This is done recursively, partitioning and repartitioning the scene to build a binary tree, where each node has an AABB, a plane which it was partitioned across, and two child nodes. Leaf nodes simply contain a list of contained primitives. My method will stop partitioning when every leaf node has no more than four primitives. This tree is a BVH – a hierarchy of bounded volumes.

To intersect with this BVH, we traverse the tree, attempting to intersect with both child AABBs at each node. For every AABB that we don't intersect with, we discard it and all its children. With this method, we are adding calculations in our AABB intersections, but generally avoiding far more calculations by discarding heaps of irrelevant primitives.

I also developed an alternative implementation of a BVH, which I called a LinearBVH (flat_bvh in the graph below). This version is functionally the exact same, but the tree is stored in a contiguous array, rather than nodes linked by pointers. My goal here was to improve memory locality.

Results

Using the included binary - `benchmarking` - I was able to create this graph by measuring the performance of the raytracer with different strategies and different numbers of objects. The tests were run on an 8 core machine, with all the 'parallel' tests using 8 threads. The scene being observed was a random collection of spheres, and every test was run five time and averaged



Since the scene was randomly generated once, then reused for all runs of each object count, I believe that many of the hitches in the graph, like the unusual uptick for 120 objects, are a result of this random generation creating a more or less favorable dataset.

You can see that paralellizing increases performance drastically, with our parallel strategies performing around 4.5x better than the non parallel ones. This is fairly far from the ideal increase of 8x, but I believe this can be accounted for by both the overhead of copying data between threads (introduced by Rust's safety measures), and the serial performance impact of encoding the image and saving it at the end of each run.

The BVH also gives a substantial performance boost, and after reading some literature, I believe there's a few tweaks I can make to my implementation to boost performance further, such as traversing each partition from front-to-back to discover our closest intersection faster.

Interestingly, my flattened BVH doesn't have any significant performance increase over the pointer-based BVH. My theory is that the Rust compiler was able to understand the binary tree pattern and apply optimizations to that, making it similarly memory efficient to my flattened tree.

Conclusion

I'm pretty pleased with this project. Ultimately I wasn't able to use MPI the way that I wanted to. My linux distribution's MPI packages didn't interface well with Rust's FFI system, so I wasn't able to get the rust_mpi bindings library to work. But I think that the complexity of my work more than makes up for that. Rust is also a fairly new language, and I think it's an absolute dream to work with, but it doesn't have many native tools for high performance computing yet, so I might have been better served in that aspect by sticking with C++.

I think that I'll be continuing work on this library going forwards, and I have a few other projects in the works in which I can leverage it. Namely, I have a Rust based website I've been working on and I think it could be cool to do some basic raytracing on the web. My first steps will be to reorganize things into a more idiomatic Rust project structure, and to expose a better thought out API for other modules to consume.