18.337 FINAL PROJECT
GPU RAY TRACING AND PLANETARY RADIATIVE TRANSFER

ZEYAD AL AWWAD [*]

Abstract. This report focuses on high-performance GPU ray tracing, implemented entirely within Julia using a custom CUDA kernel. It discusses two implementations targeting very different use cases: the first focuses on rendering 3D images, while the second focuses on simulating radiative transfer (heating/cooling) between an Earth-like planet and Sun-like star. The results include rendered images, plots of ice-albedo feedbacks in radiative transfer, and performance analysis for all of the main subcomponents (examining how the runtime scales with the number of rays and objects). A preliminary ray tracing code is available at https://github.com/Zeyad-Awwad/Julia-Ray-Tracing/tree/main, though I hope to develop this concept much further over the coming months (and since I'm just starting to prepare my thesis proposal, most likely years).

1. Introduction.

Coming from a computational physics background, and working for several years with satellite data and optical astronomy, I am particularly interested in using high-performance GPU ray tracing to study exoplanet compositions. Almost everything we can observe about other planets is optical: we can see how much light they receive, how much they absorb, how much they reflect, how atmospheres interact with the stellar spectrum, and so much more. The current leading methods for exoplanet analysis are transit methods (which focus on changes in brightness over a 3D orbit) and The complex 3D structure of exoplanets is an active area of research, but most of the spectral analysis of exoplanet observations still relies on simplified 1D models that ignore several important but challenging components (especially clouds) that might be addressed through ray tracing.

The paper is organized with the following high-level structure: the methodological details of my implementations are described in section 2, some preliminary results are shown in section 3, performance testing (with a focus on scaling) is in subsection 3.3, and some of my plans for future work that I hope to include within my thesis subsection 4.1.

2. Methodology.

2.1. Ray Tracing.

Although I developed two different models for this project (one for rendering, one for radiative transfer between a planet and star), both rely on the same basic formulation. The first section focus on general ray tracing as a computational workhorse, rather than the details of any specific implementation. It focuses on how rays are parameterized, how intersections are solved with implicit geometries, and how ray bounces are handled. The renderer uses all of these features, while the radiative transfer model only uses a subset (and adds several features that are very physics-specific).

The subsequent sections describe some features that are specific to rendering (such as color tracking and pixel averaging) and to atmospheric physics, which (even in a simple form) involve plenty of additional physical properties that must be defined, tracked and updated.

[*]PhD Candidate in Computational Science and Engineering
Engineering Systems Laboratory, Department of Aeronautics and Astronautics
MIT

2.1.1. Parameterization and Intersections.

Primary references: [3] for spherical geometries, [2] for triangular geometries.

Each ray is represented by a parameterized equation, defined by an origin $\vec{x}_0$ and a direction $\vec{d}$

$$\vec{x}(t) = \vec{x}_0 + t \cdot \vec{d}$$

This form allows us to define where a ray intersects a surface by solving for the parameter t by representing geometries with the appropriate implicit equations. The first positive intersection determines where (and from which object) the ray will bounce, and can be defined mathematically as $t_i = min\{t_{i,j} \mid t_{i,j} > 0\}$ where $t_{i,j}$ defines the intersection point between ray i and object j.

The standard approach to ray tracing uses implicit geometries, where the intersection occurs at some solution t to that equation (or set of equations). Most software uses triangular meshes to represent arbitrary geometries, but solutions to perfect spheres can also be calculated very efficiently using this approach. In practice, these two choices represent the main two geometries considered in real-world ray tracing code, but in principle, other types of geometries can be represented in similar ways.

Spheres have a relatively simple solution, since you essentially just have to solve a quadratic equation where the coefficients are given by a set of dot products between the ray and the sphere (specifically its radius $\vec{r}$ and centroid $\vec{c}$)

$$t^2(\vec{d} \cdot \vec{d}) + t(2\vec{c} \cdot \vec{d}) + \vec{c} \cdot \vec{c} = \vec{r} \cdot \vec{r}$$

This can be solved using the standard quadratic formula. Both solutions need to be checked for validity, since only real t correspond to actual intersections (complex t indicates no intersection at all) and only positive t corresponds to intersections in the direction of the ray (negative t indicates that the ray would intersect if we reverse the direction).

$$t = \frac{-(2\vec{c} \cdot \vec{d}) \pm \sqrt{(2\vec{c} \cdot \vec{d})^2 - 4(\vec{d} \cdot \vec{d})(\vec{c} \cdot \vec{c} - \vec{r} \cdot \vec{r})}}{2(\vec{d} \cdot \vec{d})}$$

Ray-triangle intersections are more computationally intensive, especially working in Cartesian coordinates (where each ray would have to solve a 3×3 system of equations). The Moller-Trumbore method is a popular method that uses a barycentric coordinate transformation to compute these intersections more efficiently.

For a triangle with vertex coordinates $\vec{A}$, $\vec{B}$ and $\vec{C}$, the transformation involves computing 3 translations with respect to vertex $\vec{A}$

$$\begin{aligned} \vec{T} &= \vec{x}_0 - \vec{A} \\ \vec{E}_1 &= \vec{B} - \vec{A} \\ \vec{E}_2 &= \vec{C} - \vec{A} \end{aligned}$$

And two cross-products that transform an arbitrary triangle into an axis-aligned unit triangle

$$\begin{aligned} \vec{P} &= \vec{d} \times \vec{E}_2 \\ \vec{Q} &= \vec{T} \times \vec{E}_1 \end{aligned}$$

This allows us to compute t with a more straightforward and GPU-friendly vector calculation

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\vec{P} \cdot \vec{E}_1} \begin{bmatrix} \vec{Q} \cdot \vec{E}_2 \\ \vec{P} \cdot \vec{T} \\ \vec{Q} \cdot \vec{D} \end{bmatrix}$$

Since t only determines when the ray intersects the plane of the triangle, we need to use the values of u and v to determine whether or not the ray is contained within the boundaries of the triangle. In Cartesian coordinates this would require a more involved point-in-polygon check, but in barycentric coordinates (where the triangle is axis-aligned with unit lengths, except for the hypotenuse) it's sufficient to check that $0 \leq u \leq 1$, $0 \leq v \leq 1$ and $0 \leq u + v \leq 1$.

Refraction is primarily handled using Snell's law: $\sin(\theta_2) = \frac{\eta_1}{\eta_2} \cdot \sin(\theta_2)$, for two materials with refractive indices η_1 and η_2. In cases where $\frac{\eta_1}{\eta_2} \cdot \sin(\theta_2) > 1$, this doesn't have a solution and we instead revert to specular reflection (physically, this represents the case of looking at water or glass surfaces from a nearly parallel angle).

2.1.2. Bounces.

Once we find the location of the first positive intersection, we update the position of the ray and decide on a new direction. The method used to determine the new direction is material-dependent and represent different physical phenomena such as specular reflections, diffuse light, refraction and thermal absorption.

Specular reflections represent an idealized "perfect mirror", where the angle of incidence (relative to the surface normal $\vec{n}$, of unit length) matches the angle of reflection. The updated direction is calculated using

$$\vec{d}_{new} = \vec{d} - 2(\vec{n} \cdot \vec{d})\vec{n}$$

Diffuse reflections draw a random new direction from some distribution. The simplest approach is to use a spherically symmetric distribution, but the Lambertian distribution is more physically accurate (since light is more likely to reflect in the perpendicular direction. Samples are drawn uniformly from the surface of a unit sphere that makes contact with the surface at the intersection point (using rejection sampling from a $2 \times 2 \times 2$ cube to find points inside the sphere, then normalizing to unit length to get points on the surface). The vector between the intersection point and the sampled surface defines the direction of the bounced ray.

2.2. Rendering.

To capture images using the traditional ray tracing code, we need to cast the rays from the camera rather than a light source. Each pixel emits the same number of rays directly perpendicular to the camera plane, with randomly sampled initial positions within the pixel boundaries. By tracking the colors of the rays, and taking the average, we can determine the color seen by each pixel.

The color of each ray is represented by an RGB vector, initially at [1.0,1.0,1.0] for all rays. With each bounce, the ray loses some color based on the material. A grassy surface, for example, may absorb 50% of the red and blue channels but leave the green channel untouched, so the bounced ray would have an updated RGB vector of [0.5, 1.0, 0.5] (and a second bounce would give [0.25, 1.0, 0.25], and so on).

A ray terminates when it flies off into the sky, the primary light source (which has a uniform blue value, and casts a slight blue tint on all visible objects). When this occurs, or when all the color values diminish too low to matter very much, their color is added to the original pixel they were cast from. This is divided by the number of rays to get an average color value and produce a full 3D-rendered image without any external API.

2.3. Radiative Transfer.

Primary references: most of this was adapted from 1D numerical integration models I implemented in the MIT course 12.815 (Atmospheric Radiation and Convection), almost entirely based on the landmark textbook "Principles of Planetary Climate" by Raymond Pierrehumbert [1].

Radiative transfer models track the heat exchange between a star and planet (considering both ground and atmospheric layers) through thermal radiation, typically after making several simplifying assumptions. For this project, we have three major simplifying assumptions

1. Everything behaves like a blackbody (a perfect absorber and emitter) with a radiation flux governed by the Stefan-Boltzmann law $F = \sigma T^4$ (given in Watts per unit area)
2. We consider only absorption/reflection of two "wavelengths": shortwave and longwave. We assume that all radiation from the star is shortwave, and all radiation from the planet is longwave. This is approximately true for the Earth and Sun, whose wavelength distributions overlap by less than 1%, and a common approach for two-stream equations in atmospheric physics, but neglects the complicated absorption profiles of real gases.
3. The planet is a tidally locked "water world" with a simple CO_2-like greenhouse atmosphere. These choices were made due to time constraints, since too many features would have taken longer to implement and evaluate. This allows us to examine the ice-albedo feedback (since the water surface can freeze/melt based on temperature), and tidal locking gives a strong temperature gradient (which is static at equilibrium, since the same side always faces the sun) with a hot side and cool side. With some small modifications, it could also be used to study the runaway greenhouse effect (but unfortunately I didn't have time to implement it before the deadline).

We will revisit these assumptions, and how they can be improved upon, when we discuss more detailed models in subsection 4.1 (Future Work). I hope to implement many of these features in my research, but my priority for this project was to become familiar with ray-traced atmospheric modeling using a relatively simple toy problem that tries to capture as much physics as possible (without missing the forest for the trees).

2.3.1. Shortwave Radiation.

Shortwave radiation comes from the star, and is primarily composed of near-infrared and visible light (as well as small but significant quantities of ultraviolet light). We can represent these by casting rays of uniform intensity from a nearby disk of radius R_{planet} representing the stellar direct beam, neglecting effects like limb darkening (where the edges of the beam aren't as bright as the center). The total incoming energy, which is evenly divided between all the rays, is based on the black-body radiation formula $F = \sigma T^4$ but also depends on the flux at that orbital distance (the ratio of the star's surface to the orbit's "surface" area) and the cross-sectional area of the planet

$$E = \frac{R_{star}^2}{R_{orbit}^2} \sigma T_{star}^4 \cdot (\pi R_{planet}^2)$$

The model assumes that the atmosphere does not interact significantly with shortwave radiation, which is a reasonable approximation for a simple atmosphere without clouds or ozone. This allows us to consider only the intersection with the planetary surface, which may absorb ($p = 1 - \alpha$) or reflect ($p = \alpha$) based on the surface albedo. Since this approach requires computing only one intersection, the shortwave step of the model is relatively fast (approximately 100 million rays per second).

Since albedo is temperature-dependent (ice is more reflective than liquid water) it can produce interesting phenomena like ice-albedo feedback, where a planet can suddenly shift from an "ice age" state (where less starlight is absorbed) to a warm state (where the surface is thawed and absorbs more starlight) depending on its initial state, due to hysteresis around unstable equilibrium points.

2.3.2. Longwave Radiation.

Since we assume a pure greenhouse atmosphere, all of the atmospheric layers (as well as the ground) can absorb and emit longwave radiation. As before, the emission is governed by the blackbody emission formula $F = \sigma T_{i,j}^4$, where $T_{i,j}$ is the local temperature at the location of emission.

The direction of emission is assumed to be spherically symmetric (as in blackbody radiation) so the emitted rays may end up being absorbed by a different location in a different layer (possibly the surface) or escaping into space. The probability of absorption decays exponentially with optical thickness τ (which measures the opacity of a path, telling us how likely it is that the gas will absorb the light) by the Beer-Lambert-Bougert law

$$I = I_0 e^{-\tau} \qquad \rightarrow \qquad p_{transmit} = e^{-\tau}, \quad p_{absorb} = 1 - e^{-\tau}$$

Using this, we can more efficiently assign an absorption layer by first drawing a probability p, then using a binary search to find the first layer that crosses that threshold (i.e. the minimum "thick enough" distance to be absorbed with this probability). This avoids needing to compute a bounce at every layer, although it still adds a significant computational expense to each longwave bounce (which takes about 6 seconds to trace 25 million rays, compared to only 2 seconds for rendering, and only a quarter of a second for shortwave).

We can keep track of the evolving temperatures with a rank-3 tensor (the programming kind, not the mathematical kind) where each layer (101 by default) is broken up into a grid (24×16 cells by default). The cells are sampled uniformly and emit a certain amount of longwave energy per time step based on their temperature and area (larger cells, near the equator, emit more total energy than smaller cells near the poles). This strikes a reasonable balance between simplicity, resolution and physical accuracy for this type of toy model, but ideally it would use a weighted sampling method that draws more rays from high-emission locations instead of compensating by scaling their energies.

2.3.3. Non-Radiative Heat Transfer.

Alternative forms of heat transfer, such as conduction and convection, are approximated using a simple weighted Jacobi method. This allows heat to diffuse to neighboring cells within the same layer (two vertical, two horizontal) by computing the average temperature and using a weighted sum to update the values

$$\tilde{T}_{i,j} = (1 - w) \cdot T_{i,j} \; + \; w \cdot \frac{T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1}}{4}$$

The weight parameter w controls the extent of diffusion. When $w = 0$, there is no temperature diffusion at all, and when $w = 1$ it gives the classical Jacobi method (where each cell is simply the average of its 4 nearest neighbors). Intermediate values are a weighted sum of these two limits, and low values of w lead to weakly diffusive behavior, which is most consistent with real tidally locked planets where we see large temperature gradients (but remains well above absolute zero, even in permanent darkness).

2.3.4. Physical and Geometric Setup.

Unlike the general ray tracer, the planetary ray tracer imposes a very restricted geometric setup for performance purposes. The planet and its atmosphere are represented by a set of concentric spheres, where the surface (layer 1) has a radius of R_{planet} and the radii of all subsequent layers are determined by the pressure profile using the pressure-altitude equation (derived from hydrostatic equilibrium)

$$R_j = R_{planet} - H \cdot \log(P_j/P_{surface})$$

Effectively, the radius increases in proportion to the log of the pressure ratio (we subtract because $P_j < P_{surface}$ for any $j > 1$, so the logarithm is always negative). The scale height is a property of the planet that represents the degree of "puffiness" of an atmosphere. It is effectively just a scaling factor for this logarithmic relationship, which is typically obtained by fitting a pressure profile to an altitude profile (I use $H = 8.5\ km$ by default, which is similar to Earth's atmosphere).

A common approach in atmospheric science is to use pressure as the primary vertical coordinate and calculate all other physical attributes from it, since it is monotonically decreasing and is more physically meaningful than altitude. In addition to the pressure-altitude relation mentioned above, we can also compute the initial temperature profile of a given layer using

$$T_j = T_{surface} \cdot (P_j/P_{surface})^{\gamma}$$

Where γ is a scaling parameter between 0 and 1. Setting it to 0.6655 reproduces the widely used 1976 US Standard Atmosphere, specifically recreating the tropospheric profile (the lowest atmospheric layer). The current model does not include any atmospheric stratification, since there is no physical mechanism to preserve those in the current model (which neglects convection and clouds). Since the purpose of this radiative transfer model is to reach thermal equilibrium, the temperature profile is not preserved when the model is run, but it provides a useful initial condition to speed up convergence and (in cases of hysteresis) potentially leads to different equilibrium points.

Similarly, we can calculate the optical thickness (opacity) profile using an almost identical scaling relationship defined by a parameter n, which is not bound to the [0,1] range. I use $n = 4$ by default, representing a fairly strong pressure-broadening effect, but other values can be physically reasonable as well

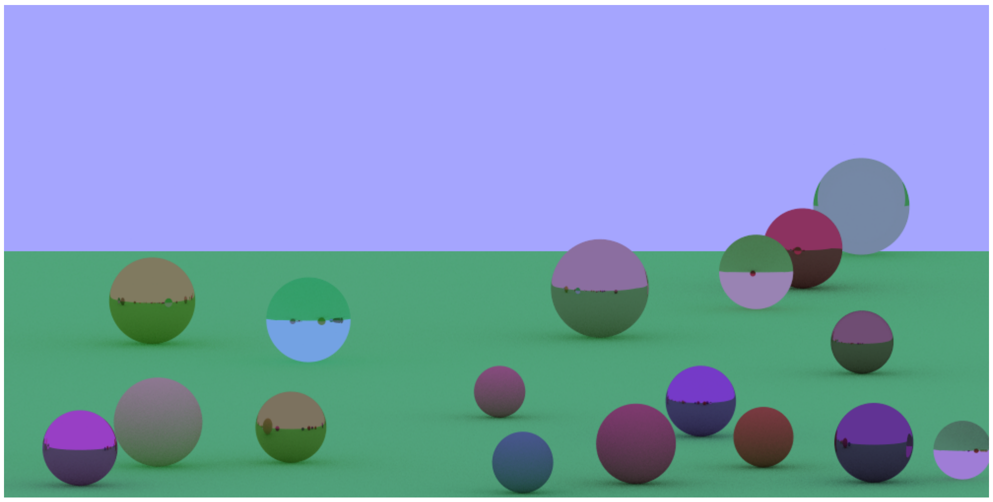
$$\tau_j = \tau_{surface} \cdot (P_j/P_{surface})^{n}$$

Since we are using a static pressure profile, the optical thickness is also static (unlike temperature). In reality, optical thickness is a complicated property that increases with both pressure and temperature. Since it also increases absorption (increasing both pressure and temperature), these interdependencies lead to sophisticated feedback loops that remain an active area of research. I hope to include that in future versions of this model, but I didn't want to tackle it this early since it would have made the scope of this project too broad.

3. Results.

3.1. Renderer.

The following image was generated using two green triangles (representing the ground) and 20 spheres of various materials and randomly assigned colors (though some may be out of frame). The colors sampling is somewhat arbitrary, but it was mostly intended to highlight certain visual effects. For example, glass spheres have a slight absorption (between 0% and 20% per channel) to make them more interesting and believable than a (completely unphysical) perfectly refracting sphere.



If you look closely, you can see several limitations in this camera-based rendering technique. Perhaps the most noticeable is that light has no direction at all, and kind of resembles a very cloudy day where light is scattered uniformly in all directions. The shadows are also rather unphysical, compared to a situation with a clear and directed light source, since they are entirely based on bounces (shaded areas occur rays cast from the camera typically can only "access the sky" after multiple bounces).

Additionally, there are a few apparent flaws that are actually correct, but admittedly still kind of weird. Perhaps the most noticeable is that highly transparent spheres don't really cast a shadow. Refracting spheres with higher absorption cast a faint shadow, but glass spheres with little absorption don't cast much of a shadow at all. This is actually not physically inaccurate, since perfect glass spheres behave in some pretty unintuitive ways. However, the lack of lensing (or any direct light source at all) makes them still look rather wrong, but some of the features in radiative transfer might help with this.

3.2. Physical Phenomena.

While I based many of the physical parameters on the Earth and Sun, this isn't a particularly realistic model and none of the numbers should be taken seriously. There is a lot of physics that needed to be ignored here to focus on high-performance GPU computation on a tight timeline. I hope to develop a physically accurate model of the future, but the goal here was to develop a ray-traced toy model that can at least demonstrate some known climate feedbacks.

3.2.1. Ice-Albedo Feedback.

The ice-albedo feedback is a phenomenon driven by an inverse relationship between albedo and temperature (i.e. ice is more reflective than water). Essentially, cold planets absorb less light and are difficult to warm, while hot planets absorb more light and are difficult to cool. This results in multiple equilibrium points for certain configurations, and it means that different initial conditions (or temporary changes in conditions) can change which equilibrium point a planet will converge to.

To demonstrate this effect, I used a fairly extreme test case where the planet begins in a frozen state (200 K across the entire surface) and is allowed to reach equilibrium before the star temporarily goes from 6000 K to 9000 K (a completely unphysical test case) to ensure the planet is knocked into another equilibrium state. I'm sure this is greater than necessary, but I didn't have enough time to test the sensitivity thoroughly. Additionally, $\alpha_{liquid} = 0.1$ and $\alpha_{ice} = 0.9$, meaning that a cold surface absorbs only 10% of incident light and a warm one absorbs 90%. These aren't too far off from the real numbers, but a bit on the extreme side.

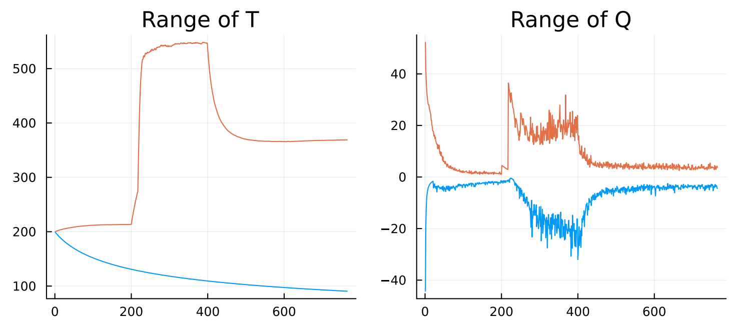


FIG. 1. *The range of temperatures (in Kelvin) and heat exchange (in Joules per timestep)* on the planetary surface. The orange line indicates the maximum value (hottest and fastest heating) and the blue line indicates the minimum value (coldest and fastest cooling)

The temperature curve shows that a planet at 200 K (completely frozen) initially approaches an equilibrium around 210 K (an ice age) but after an extreme temporary rise in temperature, it instead converges around 375 K (warmer than the Earth, but plausible for a planet without clouds near a 6000 K star). This temperature change is driven by the heat transferred Q, which measures the difference between incoming and outgoing energy for each grid cell.

Looking at Q (essentially the derivative of T), we can see that the sudden warming leads to a period of "overshooting" the equilibrium point. This is likely because the time step did not change, and may have been too coarse for such a major change in conditions. These fluctuations diminish when the star returns to normal, but they still appear to be more variable than the initial phase (starting from a frozen state).

Although the test case is quite extreme, and realistic examples of the ice-albedo feedback exhibit much smaller swings, we can see a clear example of hysteresis due to how the radiative balance changes with temperature. [Disclaimer: due to time constraints, I didn't want to test more realistic values and risk ending up with results without anything physically interesting. I also used only zero diffusion ($w = 0$) because that made hysteresis far more likely]

3.3. Performance.

The two ”versions” of the ray tracer are built on similar principles, sharing a significant amount of code, even if the overall functionality is quite different. The biggest shared DNA is in how they represent rays, how they handle spherical intersections (the most useful geometry for planetary analysis), and how they handle diffuse reflections/randomly directed emission.

However, the planetary model is more computationally intensive due to the inclusion of thermal absorption/emission and heat transfer, plus an alternate (iterative) method of determining intersections for geometries that only include concentric spheres. As a result, I cut some unnecessary features to maintain a decent thread count and avoid sacrificing too much performance.

All of my results were generated using 25 million rays per bounce, unless otherwise specified, since that fits comfortably within my card's memory (consuming approximately 14 GB in total) and takes approximately 2 seconds to compute a single rendering bounce, or a quarter of a second to compute shortwave bounce. Longwave bounces take a bit longer to complete, typically around 6 seconds for the same number of rays.

3.3.1. Rendering Performance.

To test how performance scales with the number of rays, I ran the renderer using the default settings (2 triangles forming the ground, 20 spheres with randomly assigned surface materials) using a variable number of rays and only a single bounce.

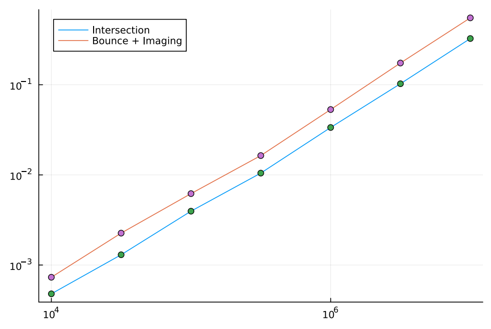


FIG. 2. *None*

We can see strong linear scaling, where 10× as many rays take approximately 10× as long. There is sometimes some fluctuation when the ray count is really small, though this unusually consistent run suggests that these are random and not caused by overhead. For the most part, the linear scaling seems to hold very well for both stages (computing intersections, and tracking bounces/pixel colors), which take approximately 0.8 and 1.2 seconds respectively for 2.5 million rays.

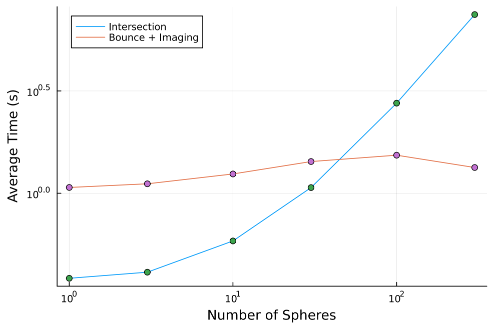


Fig. 3. *None*

When we fix the number of rays to 25 million, and instead vary the number of spheres, we can see linear scaling in computing the intersections when the number of objects is sufficiently large (e.g. 300 spheres take 10× longer than 30). This is expected due to my implementation, which checks every object for a collision (more efficient implementations use spatial queries like octrees), but implementing this on GPU is complex and well beyond the scope of this project.

Unlike the scaling with respect to rays, we see a significant amount of overhead when the number of spheres is low. This is in part due to fixed steps that every ray must execute (before and after the object loop) and because I don't vary the number of triangles (there are always 2, no matter how many spheres there are).

Additionally, although the bounce/imaging step depends on the objects, it's independent of the number of objects. Since the first intersected object was already determined in the previous step, the bounce step just needs to look up the properties of a single object (to get materials, color, etc). As a result, it remains more or less constant across all the runs, except for minor fluctuations.

3.3.2. Radiative Transfer Performance. Similar to the rendering case, we can see that GPU performance scales roughly linearly with the number of rays once there is a sufficient quantity. When the number of rays is small, there are some noticeable fluctuations in performance, but they appear to be more or less random (rather than a fixed overhead cost).

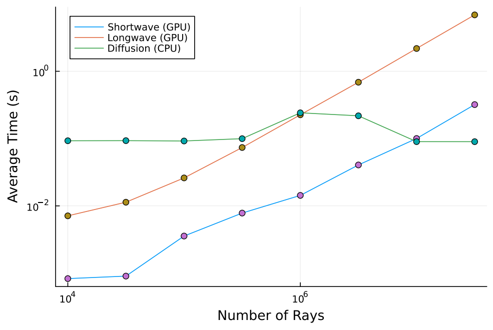


Fig. 4. *None*

The shortwave step is generally quite fast, outperforming either of the steps in the rendering case (a quarter of a second for 25 million rays). This is primarily due to the simplifications described in subsection 2.3.1. The longwave step takes significantly longer, typically around 6 seconds for 25 million rays when using a 101-layer planet/atmosphere, so I'm not sure if the "simplifications" (like the probabilistic binary search to determine the absorbing layer) are actually improving performance. This takes as long to run as the renderer's intersections with 100 spheres, though the longwave bounce both cuts and adds quite a bit so it's hard to tell (without profiling) where the performance hit is happening. I'll have to explore this further.

Additionally, the diffusion step is currently being handled entirely on CPU. It's independent of the number of rays, and (unless the number of rays is really small) has a negligible impact on performance, taking about 0.1 second for the resolution I used (slightly over 1% of the total runtime for 25 million rays). I plan to parallelize this step as well, since it would likely run faster and permit higher spatial resolutions, plus it would significantly reduce data transfer between the CPU and GPU, but it wasn't worth the effort when the longwave radiation step overwhelmingly dominates the runtime.

4. Discussion.

4.1. Future Work.

I plan to completely redo the radiative transfer form of the ray tracer over the summer, since this is really just a toy model intended to get my hands dirty with applied ray tracing in scientific computing. Unlike the renderer (which has a plethora of useful resources), this implementation was almost entirely self-guided and there is plenty of room for improvement. As a result, however, it also has a great deal of research potential (in my opinion) because current implementations are often rather limited, aimed primarily Earth's climate models rather than exoplanets, and many are closed-source commercial tools that are not easily accessible to academic researchers.

The current model neglects many important physical phenomena, and really just focuses on a simple model of thermal absorption/emission and albedo feedbacks. Real atmospheres involve many feedbacks between many of the physical parameters - for example, optical thickness/opacity is highly sensitive to both pressure and temperature, but that isn't being accounted for here. Additionally, the simple longwave/shortwave breakdown (though common in atmospheric science) is not particularly accurate, and detailed wavelength-dependent spectra are a crucial component for studying real exoplanets. Finally, the radiative impact of clouds is a complex topic that (at least in part) would greatly benefit from ray tracing, since they really can't be solved adequately without accounting for their detailed 3D structure (compared to most other atmospheric problems, which can be solved in 1D without too much loss of information).

I'm currently working on my thesis proposal, in which I hope to explore all three of these issues. The first is perhaps the hardest to truly "solve", since many of these feedback cycles involve significant amounts of uncertainty (especially regarding optical thickness and the "opacity challenge", which has received significant attention in recent years), so my goal is simply to incorporate the most widely accepted solutions into a 3D ray tracing environment (most likely something similar to petitRADTRANS).

I believe both absorption/emission spectra and cloud interactions represent major computational challenges, and research opportunities, for GPU ray tracing models. Traditional methods are quite complex and data-intensive, and typically designed for serial computing where memory (per thread) is abundant. I'm interested in exploring the potential for small neural networks to approximate these complicated, uncertain structures in an efficient, GPU-friendly way.

An additional area of improvement (not in research, just in implementation) is to combine features of both of these methods. A more realistic renderer would first model light sources and generate texture maps (similar to our temperature grid) that indicate brightness and color, then use the renderer to capture the image. The methods I added to the radiative transfer model (which only involves forward ray tracing) could be combined with the renderer (which currently only involves reverse ray tracing) to generate 3D images with more realistic lighting, as seen in current commercial tools and game engines.

Finally, I hope to get all of this working in Julia using Nvidia's Optix library, which is currently the fastest way to compute ray intersections. It relies on efficient spatial querying methods and specialized RT cores to compute 100 billion rays per second (more than 3 orders of magnitude faster than my homebrewed solution). Although that kind of work is far from my area of expertise, I think it would be a significant contribution to scientific computing as a whole, since it would make ray tracing far more accessible to academia and scientific research (because despite using C and C++ for many years, I certainly wouldn't want to develop libraries or complicated pipelines within either language).

Acknowledgments. I'd like to thank Prof. Timothy Cronin (who taught 12.815 Atmospheric Radiation and Convection in parallel to this course) for his helpful material on 1D numerical integrals of planetary atmospheres (which I tried my best to adapt somewhat faithfully into the 3D planetary ray tracing code, within the limited timeline) and his advice on how to implement some of the finer details.

REFERENCES

[1] R. Pierrehumbert, *Principles of Planetary Climate*, Cambridge University Press, 2010.
[2] P. C. Robert and D. Schweri, *GPU-based Ray-Triangle Intersection Testing*, Research Group on Computational Geometry and Graphics, Institute of Computer Science and Applied Mathematics, University of Bern, https://tr.inf.unibe.ch/pdf/iam-04-004.pdf.
[3] P. Shirley, *Ray tracing in one weekend*. https://raytracing.github.io/books/RayTracingInOneWeekend.html. Accessed: 2023-04.