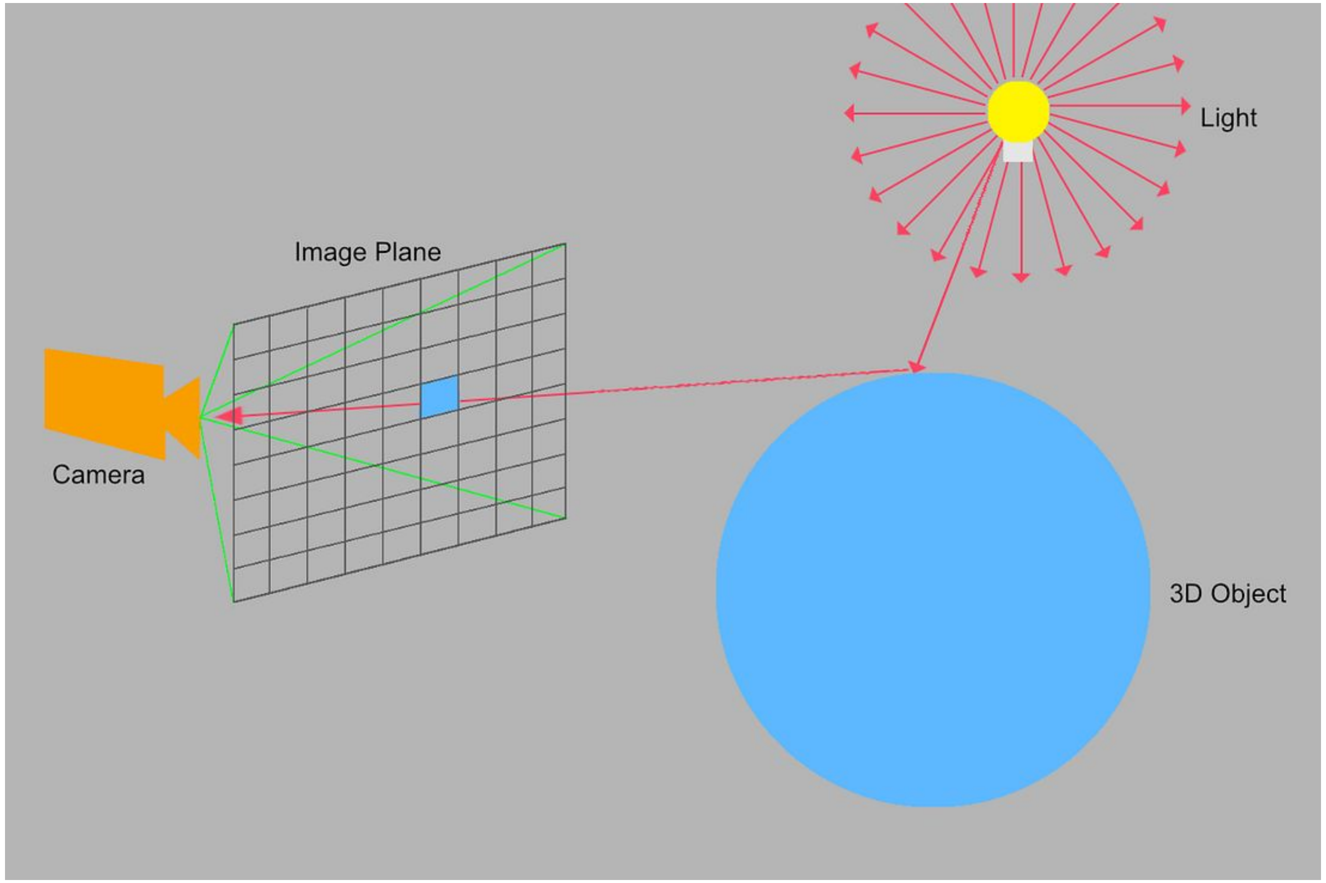
GPU-Accelerated Ray Tracing

By: Zeyad Al Awwad

Ray Tracing: Basic Idea

Track the paths of millions of light rays as they interact with the environment

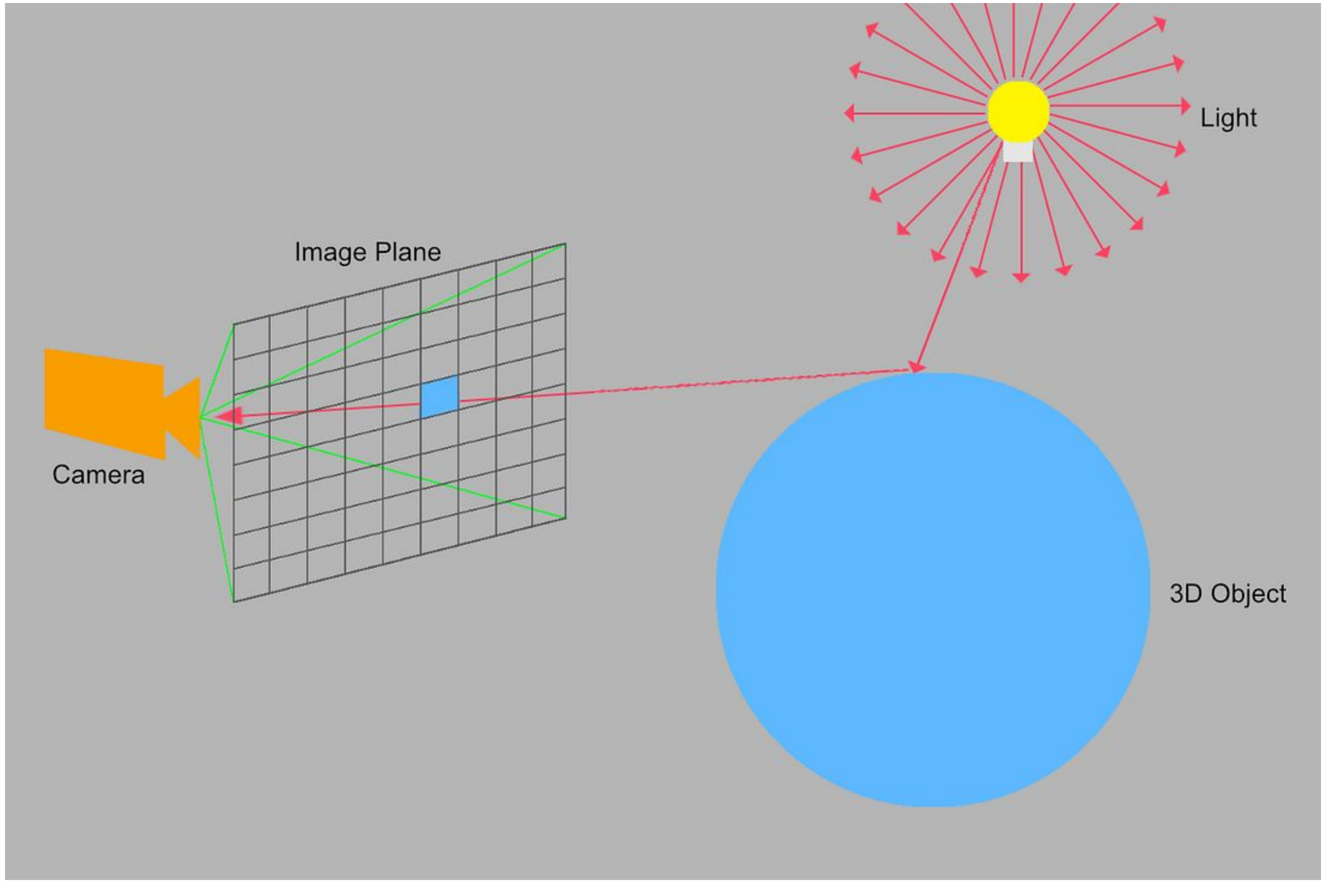


Source: https://si-ashbery.medium.com/raytracing-309fc44307e6

Ray Tracing: Basic Idea

Track the paths of millions of light rays as they interact with the environment

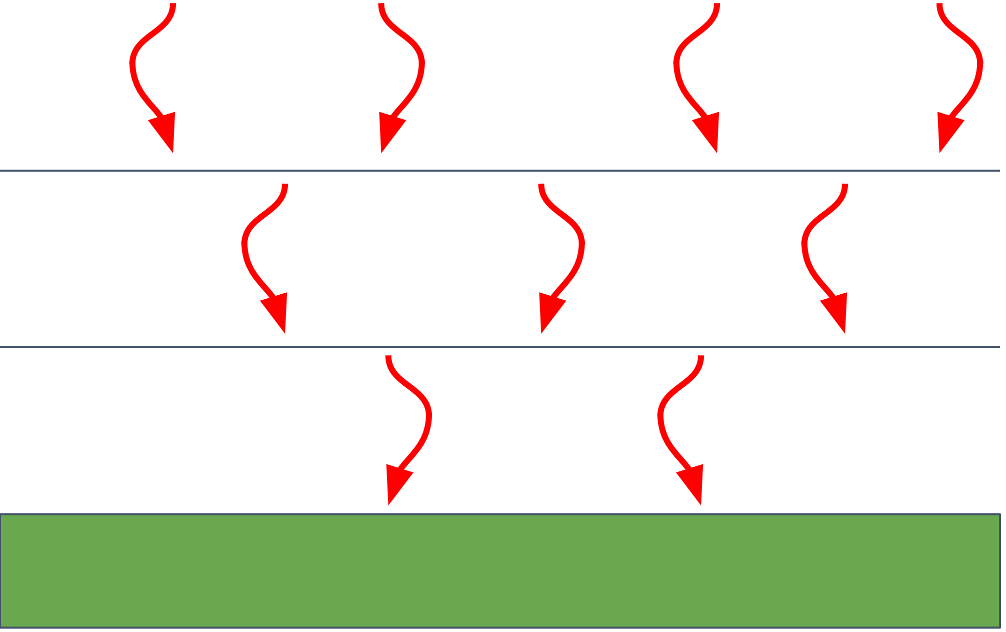
3D Renderer
Cast from the camera



Optical Simulation
Cast from light sources

Source: https://si-ashbery.medium.com/raytracing-309fc44307e6

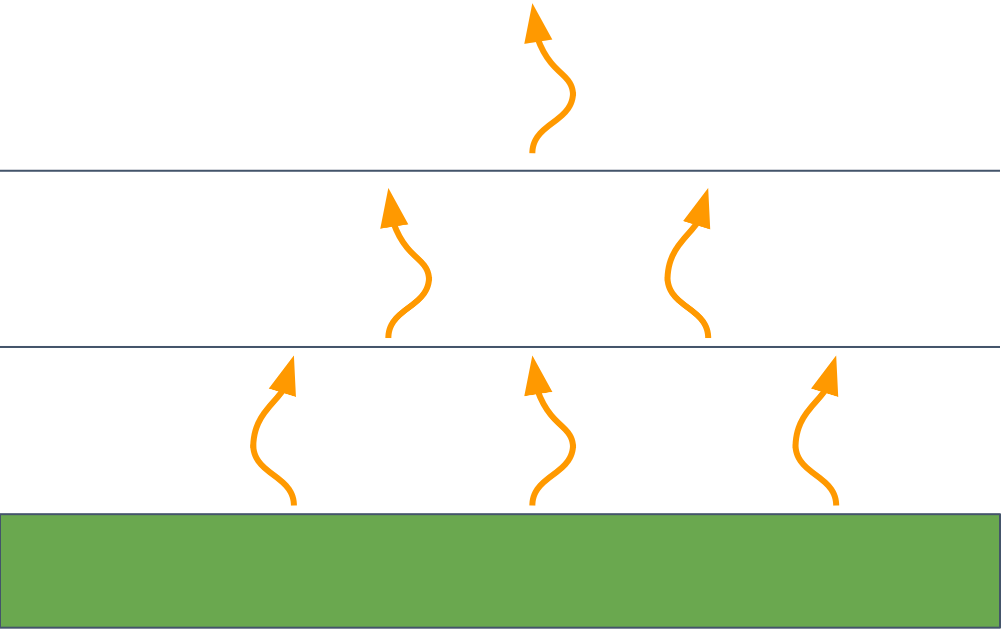
In climate modeling, we solve radiative equilibrium problems involving integrals



Every layer has a temperature, pressure, composition, opacity, etc
All coupled through radiation!
(and convection)

Scientific Applications

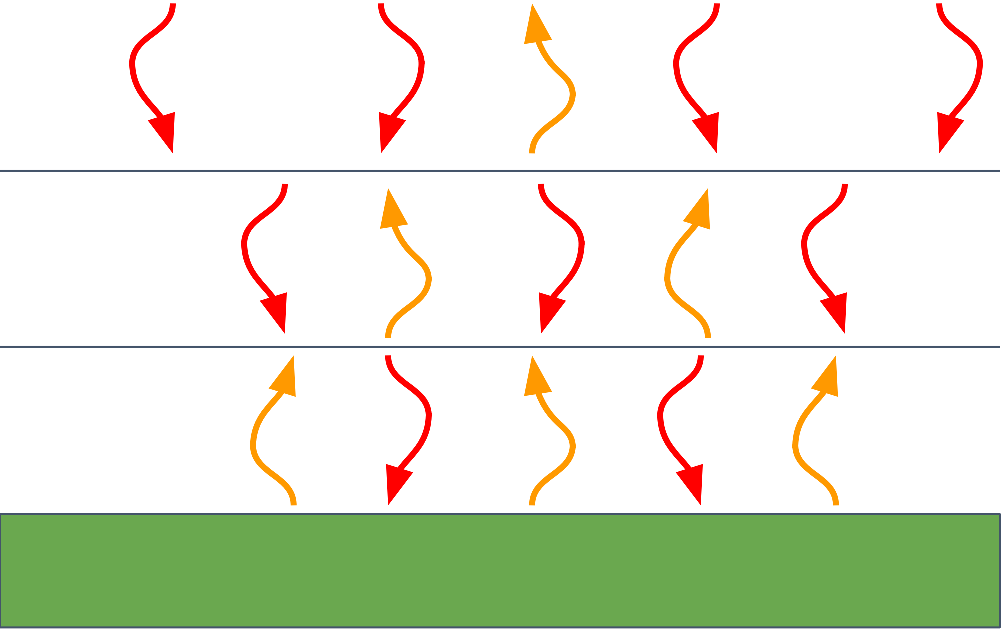
In climate modeling, we solve radiative equilibrium problems involving integrals

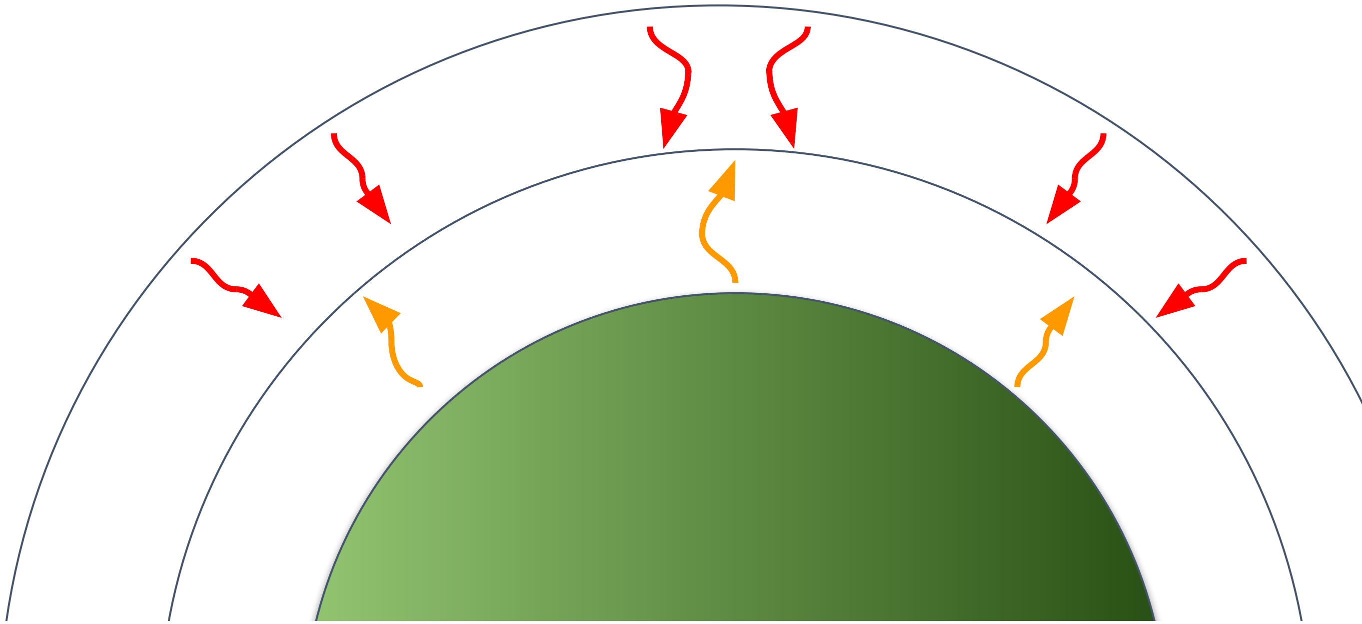


Every layer has a temperature, pressure, composition, opacity, etc
All coupled through radiation!
(and convection)

Scientific Applications

It turns out to be a scalable Monte Carlo integrator for 3D radiation problems!



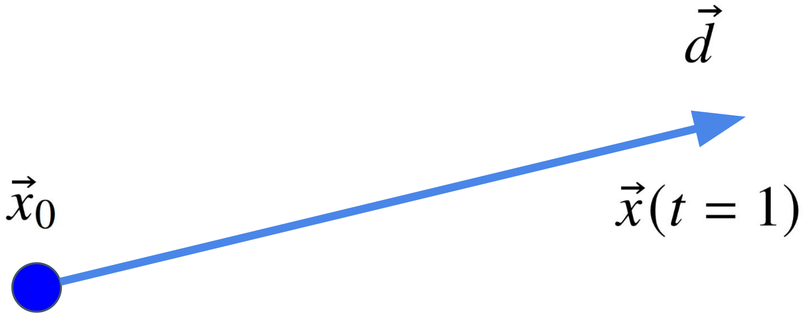


Every layer has a temperature, pressure, composition, opacity, etc
All coupled through radiation!
(and convection)

Parameterized Geometry

Light rays are modeled with a parameterized equation

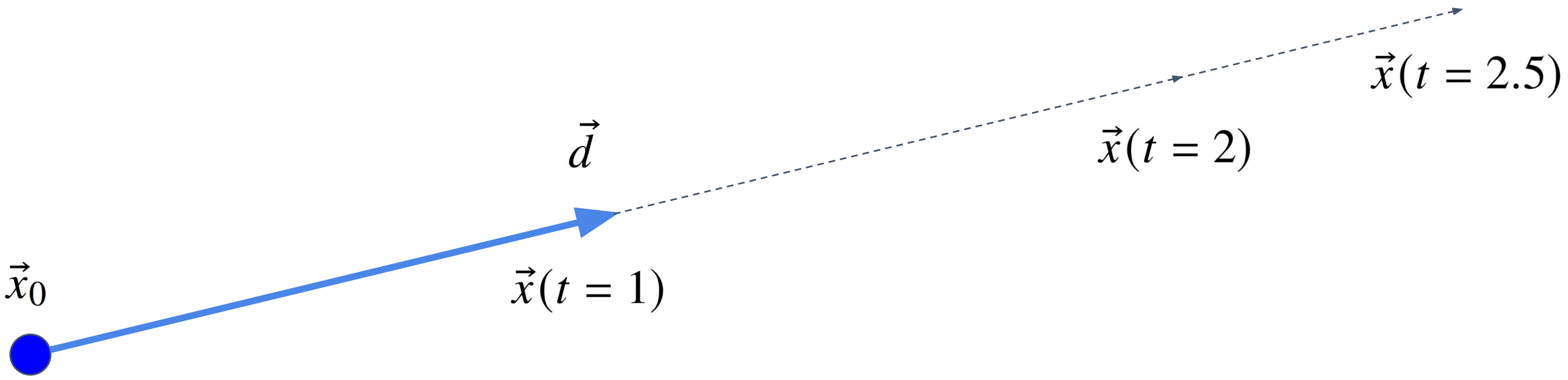
$$\vec{x}(t) = \vec{x}_0 + t \cdot \vec{d}$$



Parameterized Geometry

Light rays are modeled with a parameterized equation

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



Parameterized Geometry

Light rays are modeled with a parameterized equation

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

Shape intersections come from implicit equations, where you solve for *t*

Circles

$$t^2\ \vec{d} \cdot \vec{d} + t\ 2\vec{d} \cdot \Delta\vec{x} + \Delta\vec{x} \cdot \Delta\vec{x} = r^2$$

Triangles

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_p \\ y_a - y_p \\ z_a - z_p \end{bmatrix}$$

Being solved on every thread (need to be careful with register use!)

Light rays are modeled with a parameterized equation

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

Shape intersections come from implicit equations, where you solve for *t*

Circles

$$t^2 \boxed{\vec{d} \cdot \vec{d}} + t \boxed{2\vec{d} \cdot \Delta\vec{x}} + \boxed{\Delta\vec{x} \cdot \Delta\vec{x} = r^2}$$

$$t = \frac{b \pm \sqrt{b^2 - 4ac}}{2a}$$

Triangles

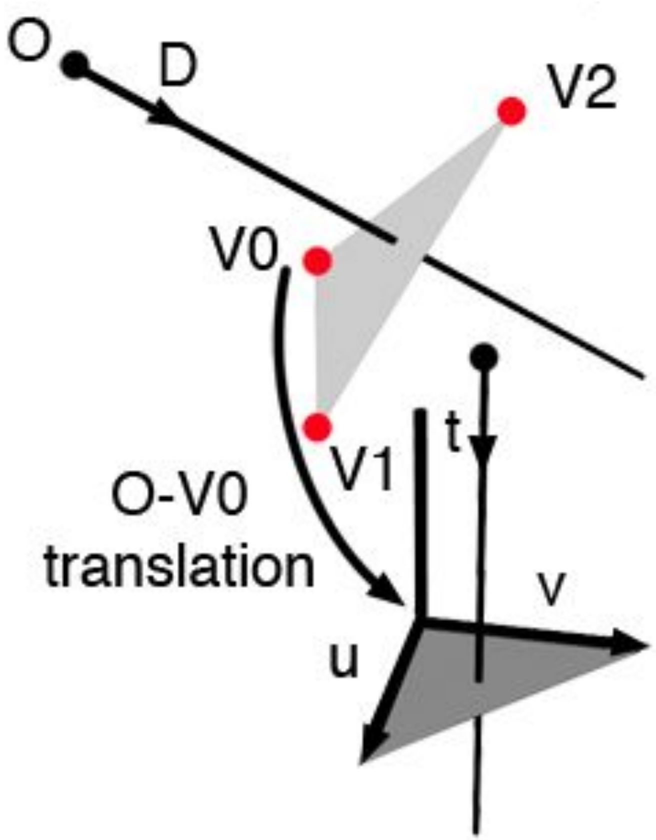
$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_p \\ y_a - y_p \\ z_a - z_p \end{bmatrix}$$

Moller-Trumbore Method

Transforming to barycentric coordinates greatly simplify triangle intersections

$$T = O - A$$
$$E_1 = B - A$$
$$E_2 = C - A$$

$$P = (D \times E_2)$$
$$Q = (T \times E_1)$$



Source: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection.html

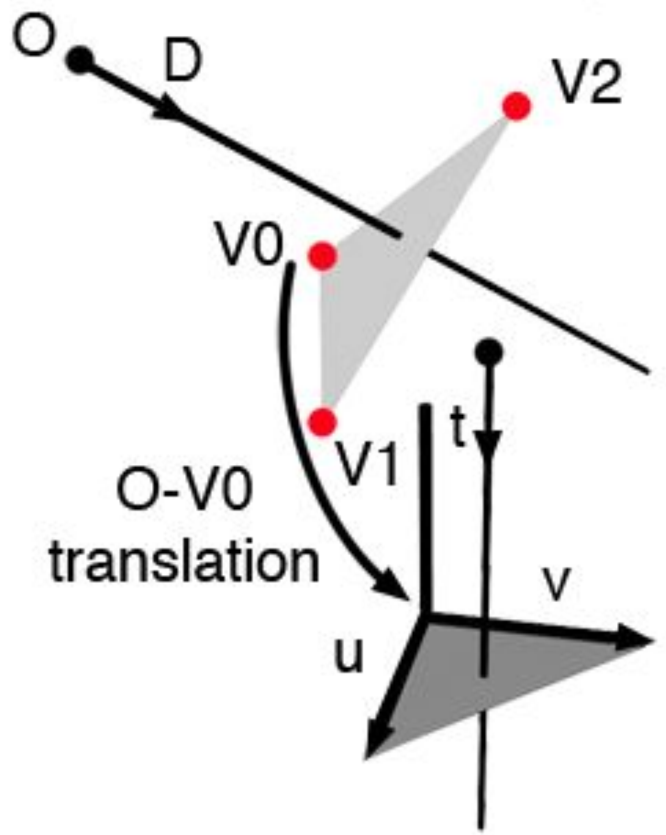
Moller-Trumbore Method

Transforming to barycentric coordinates greatly simplify triangle intersections

$$T = O - A$$
$$E_1 = B - A$$
$$E_2 = C - A$$

$$P = (D \times E_2)$$
$$Q = (T \times E_1)$$

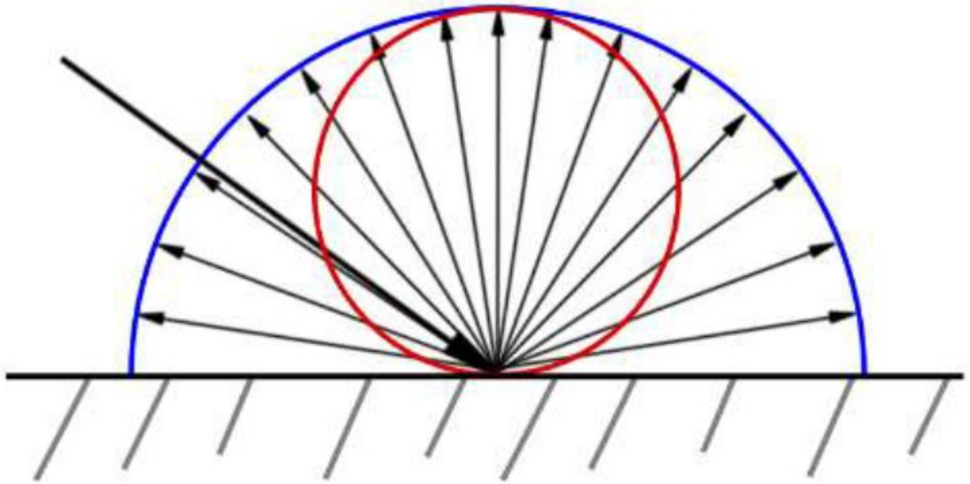
$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix}$$



Source: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection.html

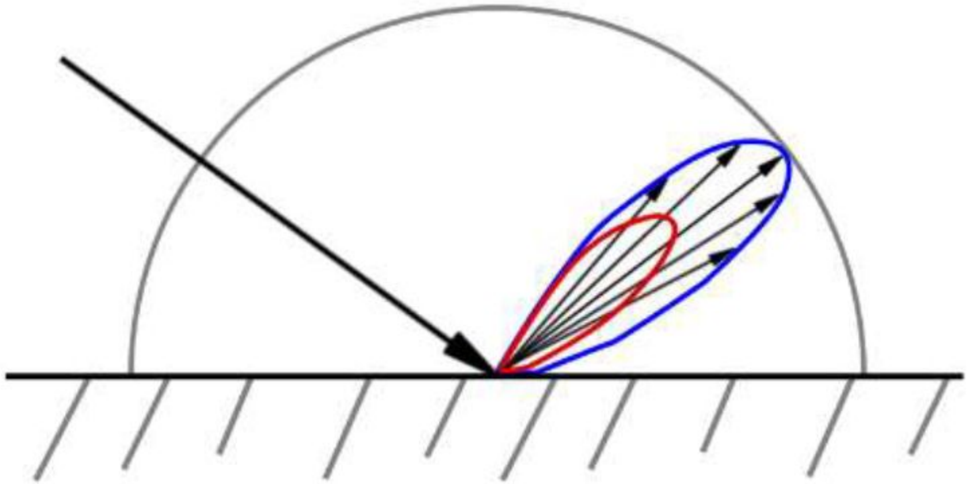
At each intersection, we may create a new ray (a “bounce”) based on effects like

Diffuse Reflections



“Matte” materials

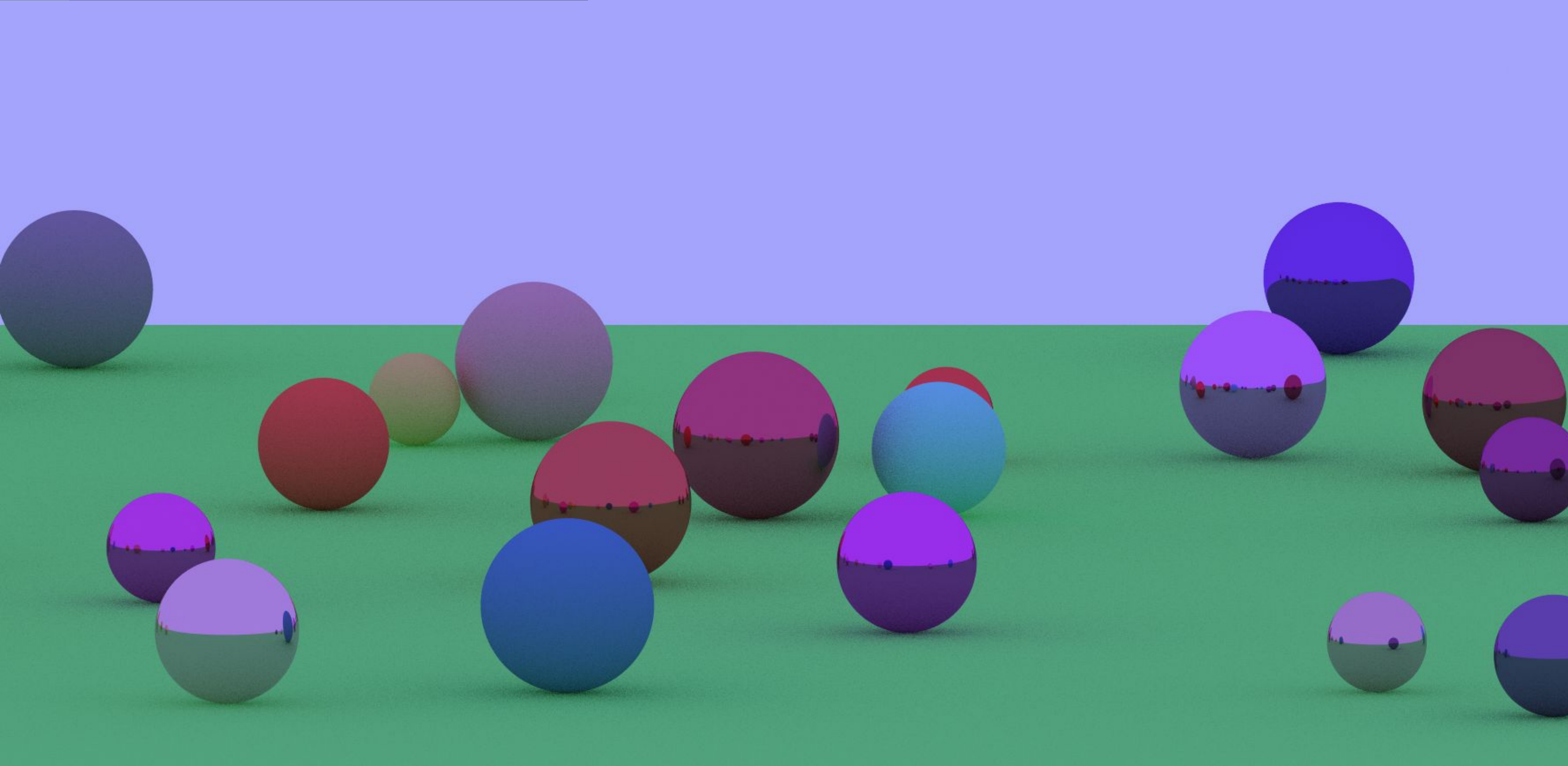
Specular Reflections



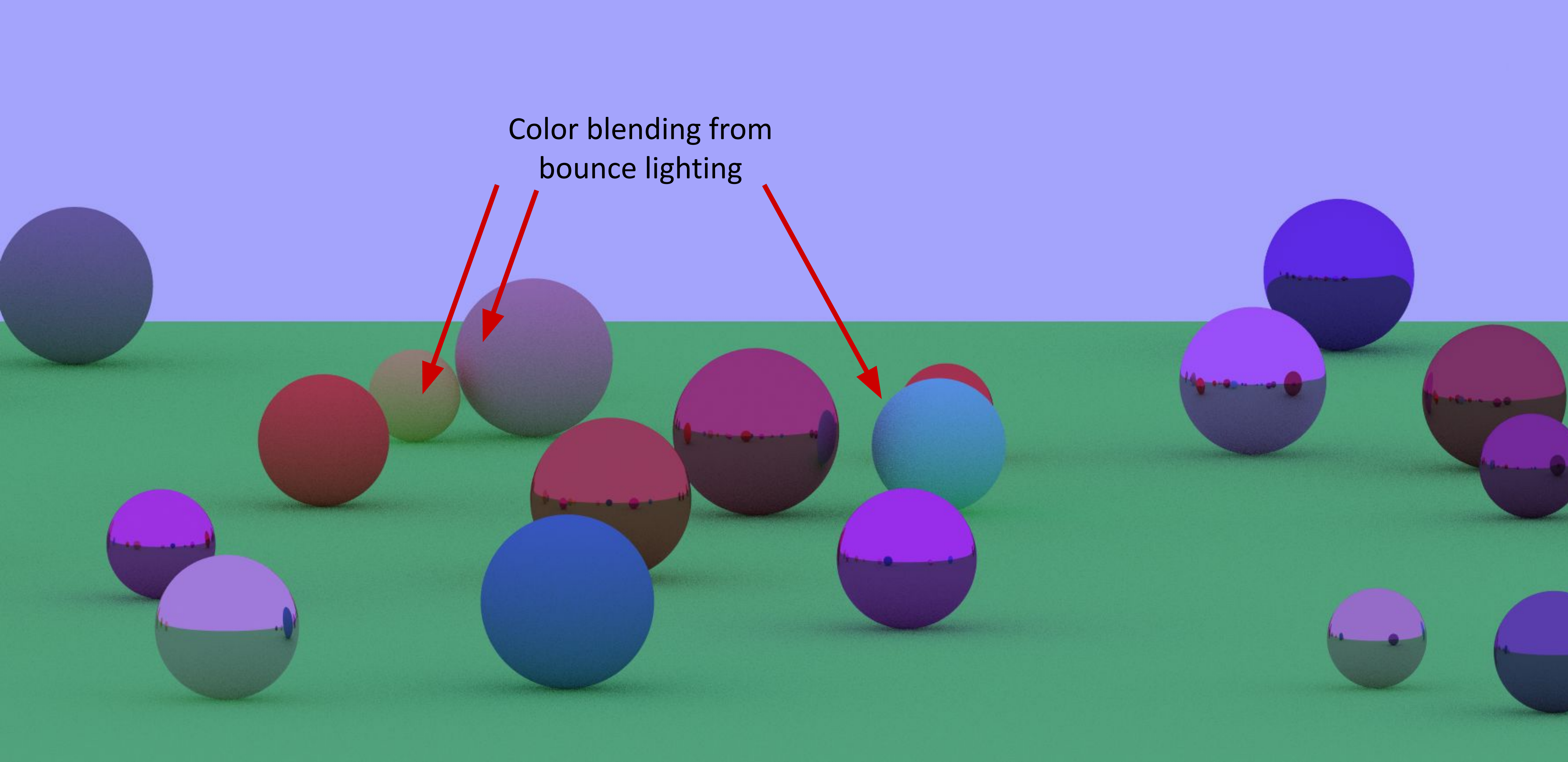
“Glossy” materials

Currently implementing refraction (transparency) and absorption (heating)

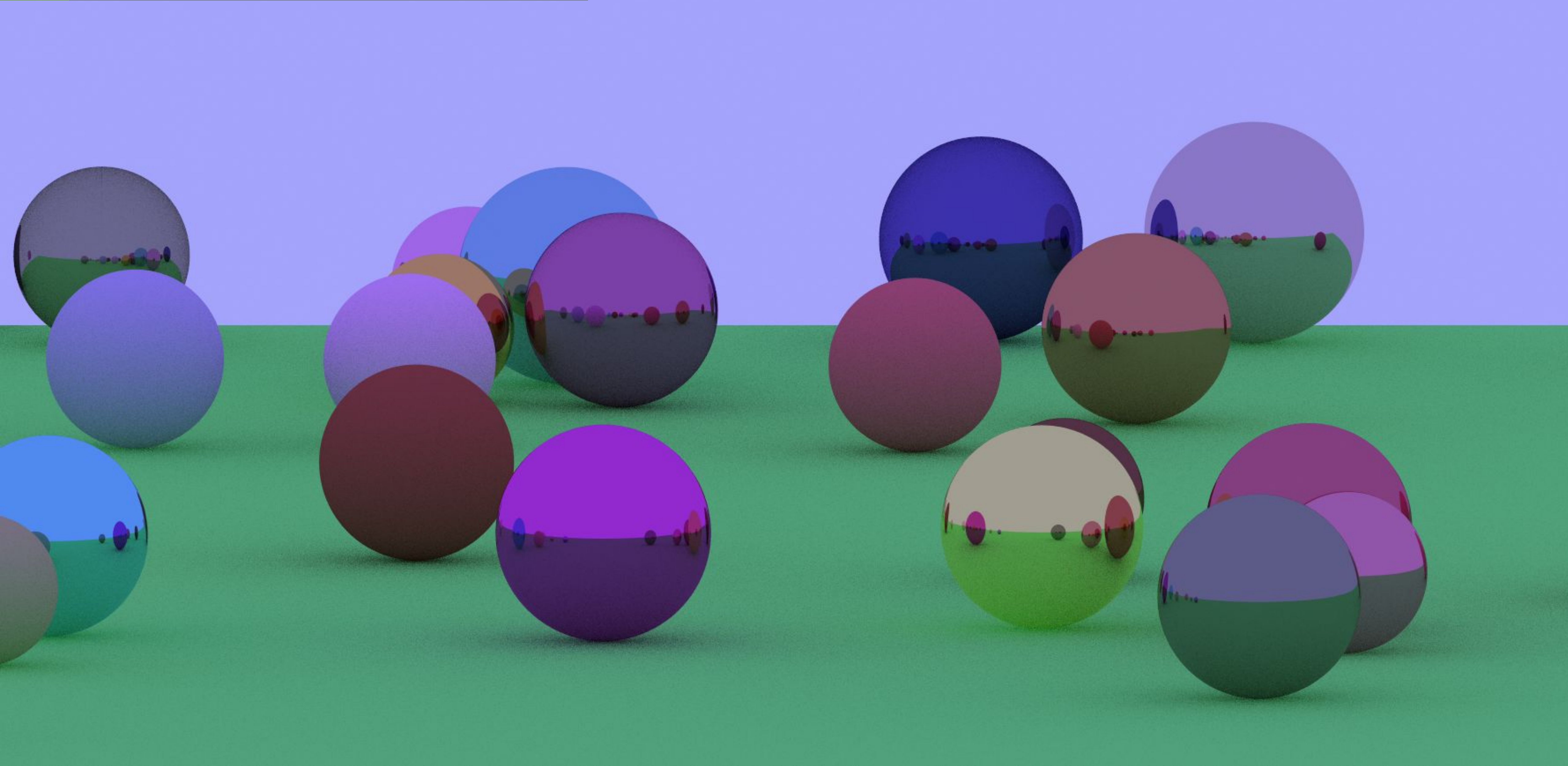
Results



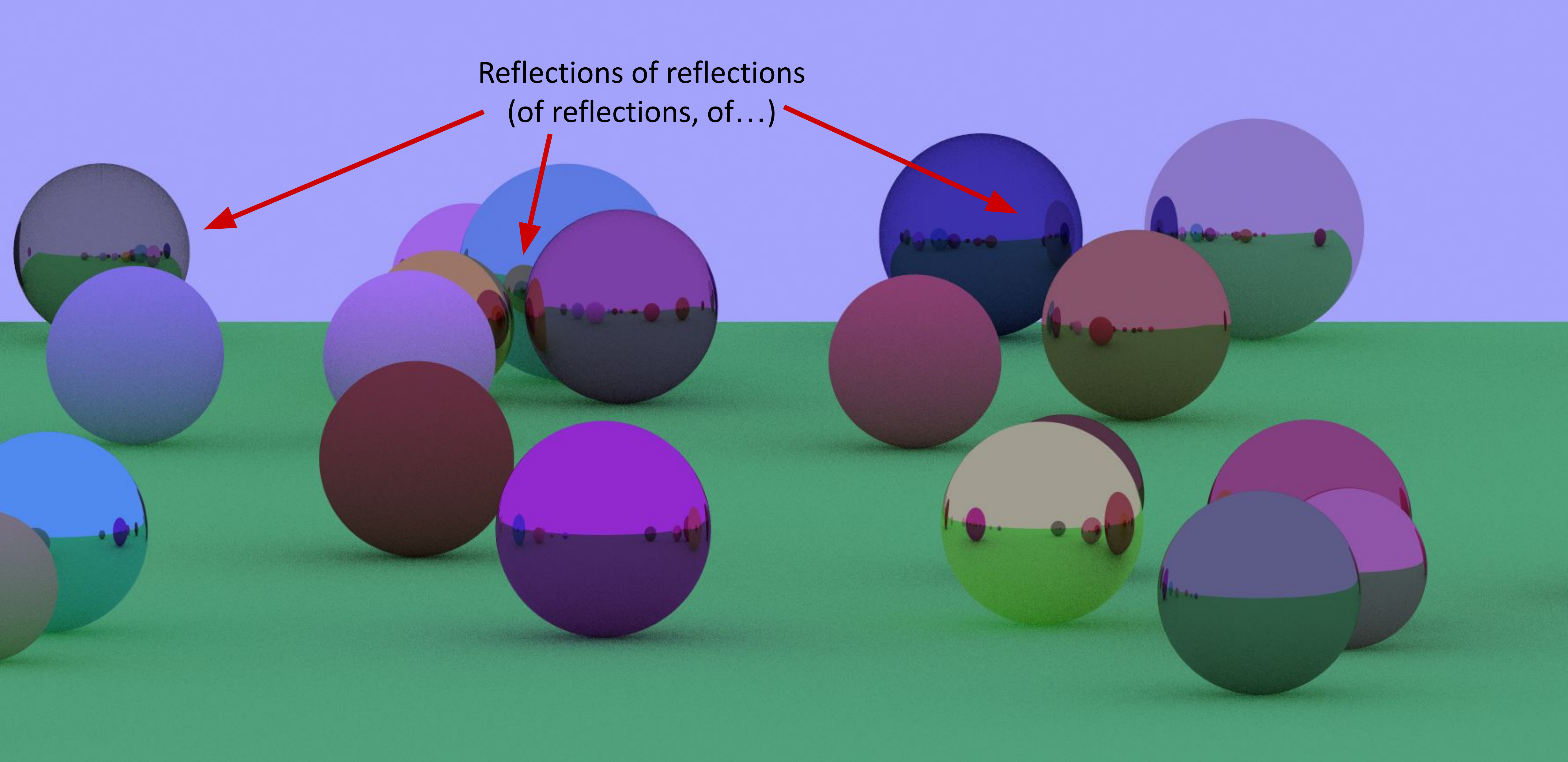
Results



Results



Results



Current Performance

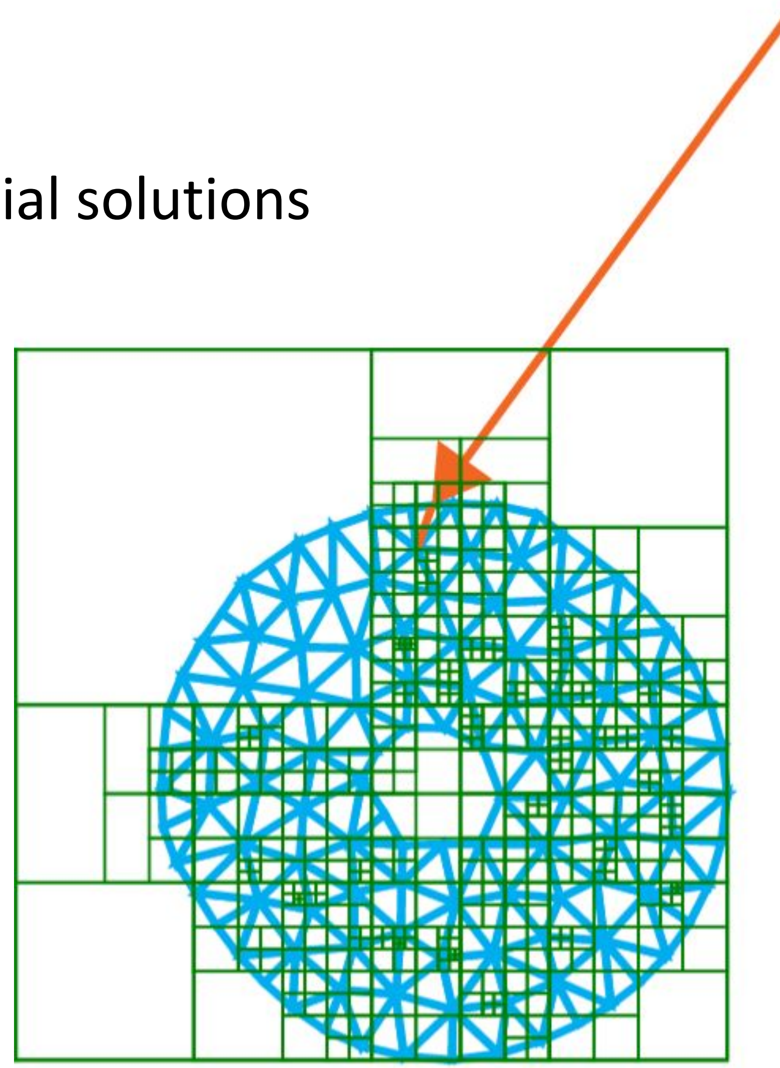
Vastly faster than CPU ray tracing, but not as fast as commercial solutions

1. Handles ~25 million rays per second
   - I gave up on my CPU implementation after a few minutes
2. Need to manage registers more carefully
   - Had to reduce thread count to 256, but I hope to get it up to 512
3. Inefficient intersections (collision detection)
   - I currently check every object, but there are much smarter solutions

Current Performance

Vastly faster than CPU ray tracing, but not as fast as commercial solutions

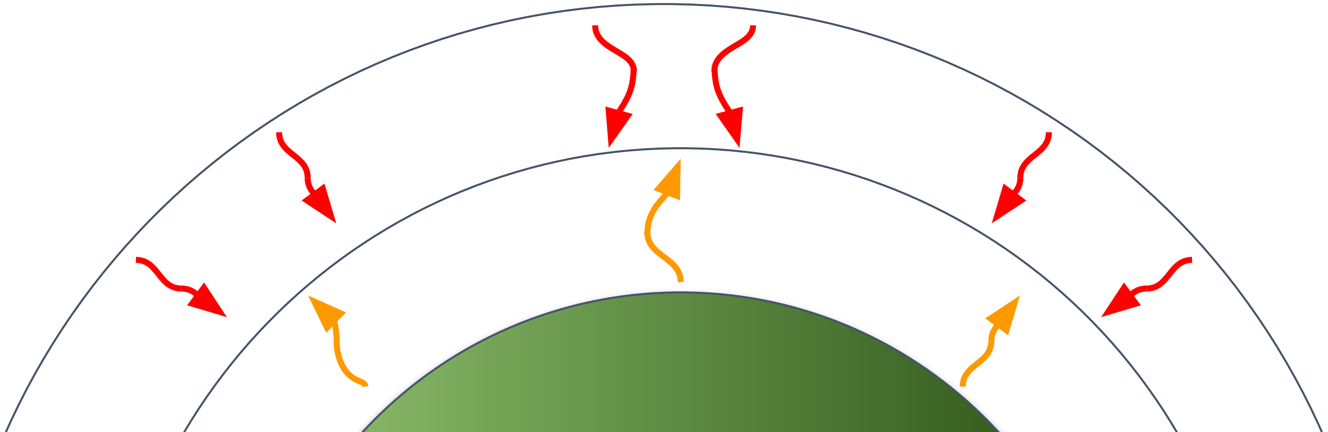
1. Handles ~25 million rays per second
   - I gave up on my CPU implementation after a few minutes
2. Need to manage registers more carefully
   - Had to reduce thread count to 256, but I hope to get it up to 512
3. Inefficient intersections (collision detection)
   - I currently check every object, but there are much smarter solutions



Short-Term Plan

In the next two weeks, I hope to

1. Add a bit more physics into the model
   - Most importantly, refraction and thermal absorption (and maybe scattering?)
2. Build an atmospheric model with this infrastructure
   - Model the radiative transfer of a simple theoretical planet
3. Examine some cool physical phenomena
   - Show how runaway greenhouse effects and ice-albedo feedbacks emerge from these models



Long-Term Plan

Over the course of my thesis, I hope to

1. Add way more physics to the model
   - 3D effects like clouds and exotic effects like molecular spectra, pressure broadening, etc
2. Switch to Nvidia's Optix library for fast intersections
   - Should give 1000x better performance for the current computational bottleneck
3. Integrate with data from data from telescopes and advanced models
   - Model exoplanets using general circulation models (GCM) and real data from stars

Call for Collaboration

If anyone is interested in getting state-of-the-art ray tracing in Julia

1. Nvidia’s Optix library is extremely well optimized
   - Uses fast collision detection and specialized hardware (RT cores)
2. Currently accessible only in C/C++
   - Fine for game engines, but a hindrance to scientific computing
3. A proof-of-concept Optix.jl interface exists
   - Not an actual library (yet) - let's make it one!