17-350 / 17-650

Lecture 06: Rust Traits and Ray Tracing

Jonathan Aldrich
Sep 15, 2026

Carnegie Mellon University

S3D Software and Societal Systems Department

Today at a Glance

- Homework 3 assigned today: safe, parallel ray tracing
- Traits: definitions, trait objects, dispatch
- Operator overloading via traits
- How a ray tracer works, and a trait-based implementation
- Demo: HW3 starter code
- Upcoming lectures: safe concurrency, then encapsulating unsafe concurrency

Next lecture: Safe Concurrency in Rust

Homework 3: A Safe, Parallel Ray Tracer

- Assigned today, due **Sep 29** (two weeks)
- Three phases, matching the next three lectures:
    i. Extend the scene with a new shape and material (**traits**, today)
    ii. Parallelize rendering, safely (**concurrency**, Thursday)
    iii. Load-balance it, with one `unsafe` feature (**next week**)
- Continues the course theme of safe abstractions with a small, well-motivated (and carefully reasoned) unsafe core
- Does not reuse HW1/HW2 code
- Short: our sample solution is under 300 lines of code

What Is a Trait?

- A trait is a set of methods a type promises to implement
- Closer to an interface than to inheritance: no shared fields, no subtyping between unrelated types

```
trait Shape {
    fn area(&self) -> f64;
}

struct Circle { radius: f64 }
struct Square { side: f64 }

impl Shape for Circle {
    fn area(&self) -> f64 { std::f64::consts::PI * self.radius * self.radius }
}

impl Shape for Square {
    fn area(&self) -> f64 { self.side * self.side }
}
```

Traits vs. Inherent `impl`

- HW1's `impl Module { fn parse_wat(...) { ... } }` is an **inherent** `impl` — methods that belong to exactly one type
- A **trait** `impl` can be written for *many* types, and other code can be written generically over "anything implementing this trait"
- Default methods: a trait method can have a body, overridable by any `impl`

```
trait Shape {
    fn area(&self) -> f64;
    fn describe(&self) -> String {          // default method
        format!("a shape with area {:.2}", self.area())
    }
}
```

`Square` inherits `describe` for free; it can still override it if it wants to.

The Problem: Heterogeneous Collections

- `Vec<Sphere>` can't also hold a `Triangle` — every element of a `Vec<T>` is the same concrete type `T`
- But HW3's scene needs exactly that: spheres, triangles, and (once you write it) triangles-with-glass, all in one list

```
// Doesn't compile: Sphere and Triangle are different types
let mut objects = vec![Sphere { .. }];
objects.push(Triangle { .. });  // error: expected `Sphere`, found `Triangle`
```

- We need a type that means "anything implementing `Hittable`," regardless of which concrete shape it actually is

The `Hittable` trait

Every shape in HW3 can answer one question: "does this ray hit you?"

```
pub trait Hittable {
    fn hit(&self, ray: &Ray, t_min: f64, t_max: f64) -> Option<HitRecord>;
}

impl Hittable for Sphere {
    fn hit(&self, ray: &Ray, t_min: f64, t_max: f64) -> Option<HitRecord> {
        // ... sphere/ray intersection math ...
    }
}
impl Hittable for Triangle {
    fn hit(&self, ray: &Ray, t_min: f64, t_max: f64) -> Option<HitRecord> {
        todo!()   // this is HW3 Phase 1
    }
}
```

- Same pattern as `Shape` / `Circle` / `Square` a few slides ago: one trait, one `impl` per concrete type
- `HitRecord` bundles up what a hit tells you (distance, point, normal, ...) — full definition in a few slides
- `Sphere` and `Triangle` are now both "hittable," even though they share no fields and no supertype

Trait Objects: `dyn Trait`

- `dyn Trait` is a way to refer to *some* type implementing `Trait`, without saying which one
- Always used behind a pointer: `&dyn Trait`, `Box<dyn Trait>`, `Arc<dyn Trait>`

```
pub struct Scene {
    pub objects: Vec<Arc<dyn Hittable>>,
}

let mut scene = Scene::new();
scene.add(Arc::new(Sphere::new(center, radius, material)));
scene.add(Arc::new(Triangle::new(p0, p1, p2, material)));
```

- One `Vec` storing heterogeneous shapes — used in HW3's `Scene`

Under the Hood: The Vtable

- While `Arc<Sphere>` is a reference counted pointer (see Thursday's lecture), `Arc<dyn Hittable>` is implemented as a reference counted *fat pointer*
  - Pairs a pointer to the object with a *vtable pointer* capturing the run-time trait implementation
  - **vtable**: a small table of function pointers for that trait's methods, e.g. specific to `Sphere`
- Calling `hittable.hit(&ray, t_min, t_max)` through a `dyn Hittable`:
  i. get the vtable pointer from the fat pointer
  ii. look up `hit`'s address in the vtable
  iii. indirect call through that function pointer
- The vtable is built once for each type that is stored in a `dyn` pointer and reused

Object safety, briefly: a trait with a generic method (`fn foo<T>(&self)`) can't become `dyn Trait` — there's no way to build one universal vtable entry for every possible `T`. Enough to recognize the error message; not a deep dive today.

Static vs. Dynamic Dispatch

Two ways to write code that works over "any `Hittable`":

```
// Static dispatch: generics + trait bounds
fn closest_hit<H: Hittable>(world: &H, ray: &Ray) -> Option<HitRecord> { ... }

// Dynamic dispatch: trait objects
fn closest_hit(world: &dyn Hittable, ray: &Ray) -> Option<HitRecord> { ... }
```

- **Generics**: the compiler *monomorphizes* — generates one specialized copy of `closest_hit` per concrete type used, resolved (and inlinable) at compile time
  - must know at some point higher in the call chain *exactly* what the type `H` is at compile time
- `dyn Trait`: one shared, non-generic function; each call is an indirect jump through the vtable. The actual type can be determined at run time.

This Is a Dispatch Tradeoff We've Seen Before

Recall Lecture 3's dispatch strategies for interpreters:

Strategy	Dispatch logic	Indirection
`match` on an enum	closed set, inlined by the compiler	branches or indirect **jump**
function table	open, via a table read at the call site	indirect **call**
`dyn Trait` vtable call	open, via a table read at the call site	indirect **call**
direct threading	open, but pre-resolved into the instruction stream	indirect **jump**

- A vtable call is structurally the same shape as Lecture 3's "function table" design: correct and flexible, with a call-boundary cost
- Generics are faster (a direct call) but less flexible at run time

Discussion: Could We Use an Enum Instead?

```rust
enum Shape {
    Sphere(Sphere),
    Triangle(Triangle),
}

impl Hittable for Shape {
    fn hit(&self, ray: &Ray, t_min: f64, t_max: f64) -> Option<HitRecord> {
        match self {
            Shape::Sphere(s) => s.hit(ray, t_min, t_max),
            Shape::Triangle(t) => t.hit(ray, t_min, t_max),
        }
    }
}

let shapes: Vec<Arc<Shape>> = ...;

for shape in &shapes {
    shape.hit(ray, t_min, t_max)
}
```

- What are the tradeoffs vs. the `dyn Hittable` solution?

Discussion: Could We Use an Enum Instead?

```
enum Shape {
    Sphere(Sphere),
    Triangle(Triangle),
}

impl Hittable for Shape {
    fn hit(&self, ray: &Ray, t_min: f64, t_max: f64) -> Option<HitRecord> {
        match self {
            Shape::Sphere(s) => s.hit(ray, t_min, t_max),
            Shape::Triangle(t) => t.hit(ray, t_min, t_max),
        }
    }
}

let shapes: Vec<Arc<Shape>> = ...;

for shape in &shapes {
    shape.hit(ray, t_min, t_max)
}
```

- The `enum` solution replaces an indirect call with an indirect jump (or branches)
- But with `enum` every new shape requires edits to this `match` (and every other `match` on `Shape`)
- `Arc<dyn Hittable>` supports **extensibility** — new shapes, even from other modules, require no changes to `Scene` or the renderer

Operator Overloading via Traits

Wouldn't it be nice to write `Vec3 + Vec3`? We can do this in a trait impl:

```
use std::ops::Add;

#[derive(Clone, Copy)]
pub struct Vec3 { pub x: f64, pub y: f64, pub z: f64 }

impl Add for Vec3 {
    type Output = Vec3;
    fn add(self, rhs: Vec3) -> Vec3 {
        Vec3 { x: self.x + rhs.x, y: self.y + rhs.y, z: self.z + rhs.z }
    }
}

let a = Vec3 { x: 1.0, y: 2.0, z: 3.0 };
let b = Vec3 { x: 0.5, y: 0.5, z: 0.5 };
let c = a + b;    // desugars to Add::add(a, b)
```

- One trait per operator, all in `std::ops`: `Add`, `Sub`, `Mul`, `Neg`, …
- Allows the starter code in `vec3.rs` to read like math notation

Two `Mul` Impls, No Conflict

HW3's `Vec3` implements `Mul` **twice**, for two different right-hand-side types — that's allowed, because they're different trait *instantiations*:

```
// Component-wise product -- used to combine material color with
// incoming light, e.g. a red surface absorbs green/blue light.
impl Mul<Vec3> for Vec3 {
    type Output = Vec3;
    fn mul(self, rhs: Vec3) -> Vec3 {
        Vec3::new(self.x * rhs.x, self.y * rhs.y, self.z * rhs.z)
    }
}

// Scalar product -- used for e.g. averaging samples.
impl Mul<f64> for Vec3 {
    type Output = Vec3;
    fn mul(self, rhs: f64) -> Vec3 {
        Vec3::new(self.x * rhs, self.y * rhs, self.z * rhs)
    }
}
```

`Mul<Vec3>` and `Mul<f64>` are as distinct to the compiler as two different traits would be — `rhs`'s type picks which one a given `*` resolves to.

How a Ray Tracer Works

For every output pixel:

1. Fire a ray from the camera through that pixel
2. Find the *closest* thing the ray hits (if anything)
3. Ask that surface what color bounces back — which may mean firing a *new* ray (a reflection or refraction) and recursing
4. If the ray hits nothing, it escapes to the sky

- Every pixel is independent of every other pixel — the reason this is a perfect setting for concurrency (Thursday's topic)
- The "ask that surface" step is a trait method call

`Hittable`, in Full

Recall `Hittable` from a few slides ago — here's the `HitRecord` it returns, and one more piece of the real signature:

```rust
pub struct HitRecord {
    pub t: f64,                 // distance along the ray
    pub p: Vec3,                // the hit point
    pub normal: Vec3,           // surface normal at the hit point
    pub front_face: bool,       // did we hit the outer surface?
    pub material: Arc<dyn Material>,
}

pub trait Hittable: Send + Sync {
    fn hit(&self, ray: &Ray, t_min: f64, t_max: f64) -> Option<HitRecord>;
}
```

- `Sphere` implements this in the starter code; `Triangle` is Phase 1
- `Send + Sync` supertraits: a scene must be safely *shareable* across every render thread (Thursday's lecture)

`Material`: "What Do You Do With Light?"

```rust
pub trait Material: Send + Sync {
    fn scatter(
        &self,
        ray_in: &Ray,
        hit: &HitRecord,
        rng: &mut SmallRng,     // for randomized scattering, with deterministic seed
    ) -> Option<(Vec3, Ray)>;   // (attenuation color, scattered ray)
}
```

- `Lambertian` (matte/diffuse) and `Metal` (mirror-like) are given
- `Dielectric` (glass-like) is Phase 1 — you implement this
- Returns `None` if the ray is fully absorbed; otherwise the color it picked up and the new ray to continue tracing

Ray Tracing and Traits

- The *set* of shapes and materials is open-ended — motivates using `dyn Trait` for its extensibility
- The renderer's core loop (`ray_color`, given in `render.rs`) never needs to know the full list of shapes or materials that exist — only that everything in the scene implements `Hittable`, and every material implements `Material`
- Adding `Triangle` and `Dielectric` requires writing exactly one `impl` block each — nothing about `Scene`, `Camera`, or the render loop changes

A Bit of Physics

- **Metal** (given): a ray reflects — angle in equals angle out, plus an optional `fuzz` for a brushed-metal look
- **Dielectric** (Phase 1): glass-like. Depending on the angle, a ray either *refracts* through the surface (bending, via Snell's law) or *reflects* off it — real glass does both, probabilistically, weighted by the angle (Schlick's approximation)
- **Exact formulas are in `hw3.md`**

Demo: Starter Code Walkthrough

[Live walkthrough] Tour of the HW3 starter project

- `vec3.rs` / `ray.rs` — the math you just saw
- `hittable.rs` — the `Hittable` trait and `HitRecord`
- `sphere.rs`, `material.rs` — given shapes/materials (`Sphere`, `Lambertian`, `Metal`); `Dielectric` stubbed with a `// TODO`
- `triangle.rs` — stubbed, `hit` always returns `None`
- `camera.rs`, `render.rs` — camera and `render_sequential`, your target API for Phase 1
- CLI: `cargo run --release -- sequential small 1 out.png`

Demo: Before and After

[Live render] The unmodified starter's small test scene

- Render it now: the glass sphere is **solid black** (`Dielectric::scatter` always returns `None` — absorbs every ray)
- The triangle is **invisible** (`Triangle::hit` always returns `None`)
- Phase 1 goal: make both of these work

Homework 3 Recap

- **Phase 1 (today's material)**: implement `Triangle::hit` and `Dielectric::scatter`, plus `refract` / `reflectance` as standalone functions — testable directly, independent of `scatter`'s randomness
- Graded by exact, hand-derivable unit tests (a ray at a triangle's centroid, `refract` at normal incidence, ...) — no rendering needed
- **Phase 2 (Thursday)**: parallelize rendering, safely
- **Phase 3 (next Tuesday)**: dynamic load balancing, and the one place this assignment needs `unsafe`

Next Lecture: Safe Concurrency in Rust

- Ray tracing is *embarrassingly parallel*: every pixel's color depends on nothing but the scene and that pixel's coordinates
- Thursday: threads, `Send` / `Sync`, and how the compiler proves your parallel code is race-free — at zero runtime cost
- By the end of Thursday's lecture, you'll have everything you need for Phase 2

Backup slide with full dyn trait Hittable example

```rust
pub trait Hittable {
    fn hit(&self, ray: &Ray, t_min: f64, t_max: f64) -> Option<HitRecord>;
}

impl Hittable for Sphere {
    fn hit(&self, ray: &Ray, t_min: f64, t_max: f64) -> Option<HitRecord> {
        // ... sphere/ray intersection math ...
    }
}

let objects: Vec<Arc<dyn Hittable>> = ...

for object in &objects {
    object.hit(ray, t_min, t_max)
}
```