Introduction

Welcome to **Linear Algebra in Rust: A Hands-On Cookbook**.

This book takes a different approach to learning linear algebra. Instead of starting with abstract theory and working toward applications, we'll jump straight into code. You'll build working programs from the first page, and the math will make sense because you'll see it in action.

Who This Book Is For

- **Programmers** who want to understand the math behind graphics, games, simulations, and machine learning
- **Students** who learn better by doing than by reading proofs
- **Rustaceans** looking to explore numerical computing
- **Anyone** curious about linear algebra but intimidated by traditional textbooks

What You'll Learn

By the end of this book, you'll be able to:

- Implement vectors, matrices, and linear transformations from scratch
- Solve systems of equations and understand when solutions exist
- Compute eigenvalues and use them for stability analysis
- Build a working 2D/3D graphics engine
- Implement machine learning algorithms using linear algebra
- Optimize numerical code for performance

What's Inside

Part I: Foundations

1. Your First Vectors
2. Matrices — Your Transformation Toolkit
3. Solving Systems of Equations
4. The Determinant and Inverse

Part II: Applications 5. Mini-Project: 2D Graphics Engine 6. Eigenvalues — Finding the Special Directions 7. Orthogonality and Projections 8. Singular Value Decomposition

Part III: Advanced Topics 9. Going 3D 10. Performance Recipes 11. Sparse Matrices 12. Iterative Solvers

Part IV: Machine Learning 13. Machine Learning Foundations 14. Capstone Projects

Appendices

- A. Rust Quick Reference
- B. Mathematical Notation

How This Book Works

Each chapter is organized as a collection of **recipes**. Every recipe follows this pattern:

1. **The Goal** — What we're building (1-2 sentences)
2. **The Math** — Just enough theory, with visual intuition
3. **The Code** — Complete, runnable Rust
4. **Try It** — Exercises to extend what you've learned
5. **Rust Note** — New language concepts when they appear

You don't need to know Rust before starting. We'll introduce language features as we need them, building from simple concepts to more advanced patterns.

Prerequisites

- Basic programming experience in any language
- High school algebra (we'll review what we need)
- Rust installed on your system (rustup.rs)

Setting Up

Create a new Rust project to follow along:

```
cargo new linear-algebra-cookbook
cd linear-algebra-cookbook
```

Each recipe includes complete code you can add to your project. By the end, you'll have built a comprehensive linear algebra library from scratch.

A Note on Notation

Throughout this book, we use standard mathematical notation:

Symbol	Meaning
$\vec{v}$	A vector
A	A matrix
$\mathbb{R}^n$	n-dimensional real space
$\|\vec{v}\|$	Length (norm) of a vector
A^T	Transpose of matrix A
A^{-1}	Inverse of matrix A

Don't worry if these are unfamiliar—we'll explain each one when we first use it.

Let's Begin

Turn the page to Chapter 1, where you'll create your first vector and start building.

No more theory. Let's code.

Chapter 1: Your First Vectors

Vectors are the fundamental building blocks of nearly everything visual and spatial in computing. Every point on your screen is described by a vector, every character in a video game moves according to velocity vectors, and every surface in a 3D model has normal vectors that determine how light bounces off it. Even machine learning, at its core, treats data as high-dimensional vectors that can be compared, clustered, and transformed.

In this chapter, you'll build a `Vec3` type from scratch and implement the fundamental operations that power computer graphics, physics simulations, and data science. By writing the code yourself, you'll develop an intuition for what these operations actually *do*—not just memorize formulas. Each recipe builds on the previous one, so by the end you'll have a complete, tested vector library that we'll use throughout the rest of the book.

What's a Vector?

At its simplest, a vector is just an ordered list of numbers. But don't let that simplicity fool you—this humble data structure is one of the most powerful abstractions in all of mathematics and computer science. The same vector can represent a position in space, a direction of travel, a force applied to an object, or a row in a database.

$$\vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

The power of vectors comes from the operations we can perform on them. When we add two velocity vectors, we get combined motion. When we compute the dot product of two vectors, we learn the angle between them. When we normalize a vector, we extract pure direction without magnitude. These operations form a consistent algebra that works regardless of what the vector represents.

Here are some concrete examples of what vectors can encode:

- **A position**: "The player is at coordinates (3, 7, 2) in the game world"
- **A direction**: "The spotlight points toward (0, -1, 0), straight down"
- **A velocity**: "The projectile moves at (5, 10, 0) meters per second"
- **Data features**: "This house has 3 bedrooms, 2 baths, and 1500 sqft"

The beauty of linear algebra is that the same math works for all of these interpretations. Learn to manipulate vectors, and you've learned to manipulate positions, directions, velocities, colors, audio samples, and data points all at once. This universality is why linear algebra appears in virtually every technical field.

What You'll Build

In this chapter, you'll implement a complete 3D vector type with all the standard operations. We start simple—just creating and printing vectors—then progressively add more sophisticated capabilities. Each recipe introduces one or two new operations along with the Rust features needed to implement them elegantly.

Recipe	What You'll Create	Key Concept
1.1	A `Vec3` struct with constructors	Structs, impl blocks, derive macros
1.2	Vector addition and subtraction	Operator overloading with traits
1.3	Scalar multiplication and division	Generic trait implementations
1.4	The dot product	Measuring angles and similarity
1.5	Length and normalization	Unit vectors and distance
1.6	The cross product	Finding perpendicular vectors

By the end, you'll have a fully-functional 3D vector type complete with tests. More importantly, you'll understand *why* each operation exists and when to use it. This foundation will make everything else in the book—matrices, transformations, eigenvalues—feel like natural extensions rather than mysterious new topics.

Rust Concepts Introduced

One of the goals of this book is to teach Rust alongside linear algebra, introducing language features as we need them. You don't need prior Rust experience, though familiarity with any programming language will help. Each recipe includes "Rust Note" sections that explain new concepts in context.

As we build our vector type, you'll learn these Rust features naturally:

- **Structs and `impl` blocks**: How to define custom types and their methods
- **Derive macros**: Automatically implementing common traits like `Debug` and `Clone`
- **Operator overloading**: Making `a + b` work for our custom types
- **Ownership and `Copy`**: Why small numeric types behave differently than strings
- **Testing**: Writing unit tests to verify our implementations

Don't worry if some of these terms are unfamiliar. We'll explain each concept when it first appears, with enough detail to understand what's happening without overwhelming you with every edge case. The goal is working knowledge, not exhaustive coverage.

Let’s start coding.

Creating Your First Vector

Recipe 1.1: Creating Your First Vector

The Goal

Before we can add, scale, or transform vectors, we need a way to represent them in code. In this recipe, we'll build a `Vec3` struct that holds three floating-point numbers and provides convenient ways to create and display vectors. This struct will be the foundation for everything else we build in this book.

Along the way, you'll learn how Rust structs work, how to add methods to them, and how derive macros can save you from writing boilerplate code. By the end, you'll have a clean, reusable vector type that feels natural to work with.

The Math

A vector in $\mathbb{R}^3$ (three-dimensional real space) is an ordered triple of real numbers. We typically write it as a column of numbers enclosed in parentheses or brackets, with the three components stacked vertically. Each component represents a coordinate along one of three perpendicular axes that we conventionally call x, y, and z.

$$\vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

We write vectors with an arrow on top ($\vec{v}$) to distinguish them from scalars (plain numbers like x). This notation helps avoid confusion in equations where both vectors and scalars appear together. Some textbooks use bold letters (**v**) instead of arrows, but the meaning is the same.

Geometric Interpretation

Think of a vector as an arrow pointing from the origin to a specific point in space. In 2D, a vector $\begin{pmatrix} 3 \\ 4 \end{pmatrix}$ points to a location 3 units right and 4 units up from the origin. You can visualize this as walking 3 steps east and 4 steps north—the vector describes both the direction and the distance of your journey.

In 3D, we simply add a third component for depth. The vector $\begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix}$ points to a location 3 units along x, 4 units along y, and 5 units along z. If you've ever used 3D modeling software or played a 3D video game, you've interacted with these coordinates, even if you didn't realize it.

The Zero Vector

One vector deserves special attention: the zero vector, which has all components equal to zero.

$$\vec{0} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

The zero vector is unique because it has no direction and no magnitude—it's just a point sitting at the origin. Geometrically, it represents "no displacement" or "staying in place." Algebraically, it serves as the identity element for vector addition: adding the zero vector to any vector leaves it unchanged. We'll see this property in action in Recipe 1.2.

The Code

Let's translate these mathematical concepts into Rust. Create a new file called `src/vec3.rs` in your project. This module will contain our vector type and all its associated methods.

```rust
// src/vec3.rs

/// A 3-dimensional vector with f64 components.
///
/// Vec3 represents a point or direction in 3D space. It's the fundamental
/// building block for graphics, physics, and geometric computations.
#[derive(Debug, Clone, Copy, PartialEq)]
pub struct Vec3 {
    pub x: f64,
    pub y: f64,
    pub z: f64,
}

impl Vec3 {
    /// Creates a new Vec3 with the given components.
    ///
    /// This is the most common way to create a vector when you know
    /// all three coordinates upfront.
    pub fn new(x: f64, y: f64, z: f64) -> Self {
        Vec3 { x, y, z }
    }

    /// Creates the zero vector (0, 0, 0).
    ///
    /// The zero vector represents no displacement and serves as the
    /// identity element for vector addition.
    pub fn zero() -> Self {
        Vec3 { x: 0.0, y: 0.0, z: 0.0 }
    }

    /// Creates a unit vector along the X axis: (1, 0, 0).
    ///
    /// This is one of the three standard basis vectors that span 3D space.
    /// Any vector can be expressed as a combination of unit_x, unit_y, and
unit_z.
    pub fn unit_x() -> Self {
        Vec3 { x: 1.0, y: 0.0, z: 0.0 }
    }

    /// Creates a unit vector along the Y axis: (0, 1, 0).
    pub fn unit_y() -> Self {
        Vec3 { x: 0.0, y: 1.0, z: 0.0 }
    }

    /// Creates a unit vector along the Z axis: (0, 0, 1).
    pub fn unit_z() -> Self {
        Vec3 { x: 0.0, y: 0.0, z: 1.0 }
    }
}

// Implement Display for pretty printing
use std::fmt;

impl fmt::Display for Vec3 {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "({}, {}, {})", self.x, self.y, self.z)
```

```
    }
}
```

Now update `src/main.rs` to use our new module. This file serves as the entry point for our program and lets us test our vector type interactively.

```
// src/main.rs

mod vec3;
use vec3::Vec3;

fn main() {
    // Create vectors using different constructor methods.
    // Each method serves a different purpose, making code more readable.
    let v1 = Vec3::new(1.0, 2.0, 3.0);
    let v2 = Vec3::zero();
    let v3 = Vec3::unit_x();

    // Debug output shows the full struct with field names.
    // This is useful during development and debugging.
    println!("Debug: {:?}", v1);

    // Display output is cleaner and meant for end users.
    // We implemented this ourselves with the Display trait.
    println!("v1 = {}", v1);
    println!("v2 = {}", v2);
    println!("v3 = {}", v3);

    // We can access individual components directly since they're public.
    // This is useful when you need just one coordinate.
    println!("v1.x = {}", v1.x);
    println!("v1.y = {}", v1.y);
    println!("v1.z = {}", v1.z);
}
```

Run the program with `cargo run` to see the output:

```
Debug: Vec3 { x: 1.0, y: 2.0, z: 3.0 }
v1 = (1, 2, 3)
v2 = (0, 0, 0)
v3 = (1, 0, 0)
v1.x = 1
v1.y = 2
v1.z = 3
```

Rust Note: Derive Macros

You may have noticed the `#[derive(...)]` attribute above our struct definition. This is one of Rust's most convenient features—it automatically generates implementations of

common traits based on the struct's fields. Without derive macros, we'd have to write dozens of lines of boilerplate code ourselves.

```
#[derive(Debug, Clone, Copy, PartialEq)]
pub struct Vec3 { ... }
```

Each derived trait gives our struct new capabilities:

Trait	What it does	Why we want it
`Debug`	Enables `{:?}` formatting	Print vectors during debugging
`Clone`	Allows explicit `.clone()` copies	Duplicate vectors when needed
`Copy`	Enables implicit copying	Pass vectors without ownership transfer
`PartialEq`	Enables `==` and `!=`	Compare vectors for equality

The `Copy` trait deserves special attention because it fundamentally changes how Rust handles our type. Normally, when you assign one variable to another in Rust, ownership moves and the original becomes invalid. But `Copy` types are implicitly duplicated instead, behaving like primitive numbers.

```
let a = Vec3::new(1.0, 2.0, 3.0);
let b = a;  // 'a' is copied, not moved!
println!("{}", a);  // This works because 'a' still exists

// Compare to String, which doesn't implement Copy:
let s1 = String::from("hello");
let s2 = s1;  // s1 is moved, not copied
// println!("{}", s1);  // ERROR: s1 no longer valid
```

We can safely implement `Copy` for `Vec3` because it's small (just 24 bytes—three 8-byte floats) and contains no heap allocations. Copying it is cheap and has no side effects, which are the requirements for `Copy` types.

Rust Note: `Self` vs `Vec3`

Inside an `impl Vec3` block, the keyword `Self` serves as an alias for `Vec3`. You can use either one, but `Self` has advantages: it's shorter to type, and if you ever rename the struct, your return types and constructor calls update automatically.

```
impl Vec3 {
    // Both of these are equivalent:
    pub fn new(x: f64, y: f64, z: f64) -> Self {
        Vec3 { x, y, z }
    }

    pub fn new_alt(x: f64, y: f64, z: f64) -> Vec3 {
        Vec3 { x, y, z }
    }
}
```

Most Rust developers prefer `Self` for return types within impl blocks. It's a small thing, but these conventions add up to more maintainable code.

Try It

Now that you have a working vector type, try extending it with these exercises. Each one teaches a useful pattern that you'll use throughout the book.

1. **Add a constructor for 2D vectors.** Many games and applications work in 2D, so it's convenient to have a constructor that assumes z = 0:

   ```
   pub fn new_2d(x: f64, y: f64) -> Self {
       Vec3 { x, y, z: 0.0 }
   }
   ```

2. **Create vectors from arrays.** Sometimes you'll receive vector data as arrays from external sources. Add conversion methods in both directions:

   ```
   pub fn from_array(arr: [f64; 3]) -> Self {
       Vec3 { x: arr[0], y: arr[1], z: arr[2] }
   }

   pub fn to_array(&self) -> [f64; 3] {
       [self.x, self.y, self.z]
   }
   ```

3. **Add negation.** Negating a vector reverses its direction. We'll implement proper operator overloading for `-v` in Recipe 1.2, but for now, add a method:

   ```
   pub fn negate(&self) -> Self {
       Vec3 { x: -self.x, y: -self.y, z: -self.z }
   }
   ```

4. **Write your first tests.** Rust has built-in support for unit testing. Add a tests module at the bottom of `vec3.rs`:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_new() {
        let v = Vec3::new(1.0, 2.0, 3.0);
        assert_eq!(v.x, 1.0);
        assert_eq!(v.y, 2.0);
        assert_eq!(v.z, 3.0);
    }

    #[test]
    fn test_zero() {
        let v = Vec3::zero();
        assert_eq!(v, Vec3::new(0.0, 0.0, 0.0));
    }
}
```

   Run the tests with `cargo test`. You should see both tests pass. As we add more functionality, we'll add more tests to ensure everything works correctly.

What's Next?

You now have vectors, but they just sit there—we can create them and print them, but we can't combine them or transform them. In Recipe 1.2, we'll bring them to life with addition and subtraction. Along the way, you'll learn Rust's powerful operator overloading system, which lets us write `a + b` instead of `a.add(b)`.

Adding Vectors

Recipe 1.2: Adding Vectors

The Goal

With our `Vec3` struct in place, it's time to make vectors do something useful. Vector addition is the most fundamental operation—it's how we combine displacements, accumulate velocities, and blend colors. By the end of this recipe, you'll be able to write `a + b` with the same natural syntax you'd use for numbers.

This recipe also introduces one of Rust's most elegant features: operator overloading through traits. Instead of calling awkward methods like `a.add(b)`, we'll implement the `Add` trait so that the `+` operator just works. This pattern will serve us well throughout the book as we add more operations to our vector type.

The Math

Vector addition is beautifully simple: just add the corresponding components. Take the x-components and add them together, do the same for y and z, and you have your result. There's no complicated formula to memorize—it's exactly what you'd expect.

$$\vec{a} + \vec{b} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} + \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} a_x + b_x \\ a_y + b_y \\ a_z + b_z \end{pmatrix}$$

Let's work through a concrete example to make sure the concept is clear:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 1 + 4 \\ 2 + 5 \\ 3 + 6 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \\ 9 \end{pmatrix}$$

Geometric Interpretation

While the arithmetic is straightforward, the geometric meaning is what makes vectors powerful. Imagine you're walking through a city on a grid. Vector $\vec{a}$ takes you 3 blocks east

and 2 blocks north. Vector $\vec{b}$ takes you 1 block east and 4 blocks north. If you walk $\vec{a}$ first, then $\vec{b}$, where do you end up?

The answer is $\vec{a} + \vec{b}$: 4 blocks east and 6 blocks north from your starting point. This is sometimes called the **parallelogram rule** because if you draw both vectors from the same starting point, their sum is the diagonal of the parallelogram they form.

Mathematical Properties

Vector addition satisfies several important properties that you might recognize from regular number addition. These properties aren't just mathematical curiosities—they're guarantees that make vector math predictable and trustworthy.

Property	Formula	What It Means
Commutative	$\vec{a} + \vec{b} = \vec{b} + \vec{a}$	The order doesn't matter
Associative	$(\vec{a} + \vec{b}) + \vec{c} = \vec{a} + (\vec{b} + \vec{c})$	Grouping doesn't matter
Identity	$\vec{a} + \vec{0} = \vec{a}$	Adding zero changes nothing
Inverse	$\vec{a} + (-\vec{a}) = \vec{0}$	Every vector has an opposite

The commutative property means you can add vectors in any order. The associative property means you can group additions however you like. These might seem obvious, but not all mathematical operations have these properties (matrix multiplication, for instance, is not commutative).

Vector Subtraction

Once we have addition and negation, subtraction comes for free. Subtracting a vector is the same as adding its negation:

$$\vec{a} - \vec{b} = \vec{a} + (-\vec{b}) = \begin{pmatrix} a_x - b_x \\ a_y - b_y \\ a_z - b_z \end{pmatrix}$$

Geometrically, $\vec{a} - \vec{b}$ gives you the vector pointing from the tip of $\vec{b}$ to the tip of $\vec{a}$. This is incredibly useful: if you have two positions and want to know the direction from one to the other, just subtract them.

The Code

Now let's implement these operations in Rust. We'll use operator overloading, which means implementing special traits from the standard library. Add these imports and trait implementations to `src/vec3.rs`:

```
// Add these imports at the top of vec3.rs, after any existing imports
use std::ops::{Add, Sub, Neg};

// Implement the Add trait to enable the + operator.
// When Rust sees "a + b" where both are Vec3, it calls this method.
impl Add for Vec3 {
    type Output = Vec3;

    fn add(self, other: Vec3) -> Vec3 {
        Vec3 {
            x: self.x + other.x,
            y: self.y + other.y,
            z: self.z + other.z,
        }
    }
}

// Implement the Sub trait to enable the - operator (binary minus).
// Subtraction follows the same pattern as addition.
impl Sub for Vec3 {
    type Output = Vec3;

    fn sub(self, other: Vec3) -> Vec3 {
        Vec3 {
            x: self.x - other.x,
            y: self.y - other.y,
            z: self.z - other.z,
        }
    }
}

// Implement the Neg trait to enable the unary - operator.
// This is the minus sign with no left operand, as in "-a".
impl Neg for Vec3 {
    type Output = Vec3;

    fn neg(self) -> Vec3 {
        Vec3 {
            x: -self.x,
            y: -self.y,
            z: -self.z,
        }
    }
}
```

Now you can use natural mathematical syntax in `main.rs`. The code reads almost like the mathematical formulas we wrote earlier:

```rust
fn main() {
    let a = Vec3::new(1.0, 2.0, 3.0);
    let b = Vec3::new(4.0, 5.0, 6.0);

    // Addition works just like with numbers
    let sum = a + b;
    println!("a + b = {}", sum);  // (5, 7, 9)

    // Subtraction too
    let diff = a - b;
    println!("a - b = {}", diff);  // (-3, -3, -3)

    // Negation reverses all components
    let neg_a = -a;
    println!("-a = {}", neg_a);  // (-1, -2, -3)

    // You can chain operations naturally
    let result = a + b - Vec3::unit_x();
    println!("a + b - unit_x = {}", result);  // (4, 7, 9)

    // Let's verify those mathematical properties we discussed
    let zero = Vec3::zero();
    assert_eq!(a + zero, a);
    println!("Identity property verified: a + 0 = a");

    assert_eq!(a + (-a), Vec3::zero());
    println!("Inverse property verified: a + (-a) = 0");
}
```

The output confirms our implementations match the mathematical definitions:

```
a + b = (5, 7, 9)
a - b = (-3, -3, -3)
-a = (-1, -2, -3)
a + b - unit_x = (4, 7, 9)
Identity property verified: a + 0 = a
Inverse property verified: a + (-a) = 0
```

Rust Note: Operator Overloading

Rust doesn't have magical operator overloading like some languages. Instead, operators are defined by traits in the `std::ops` module. When you write `a + b`, Rust looks for an implementation of `Add` for the types involved. This design is more explicit and predictable than implicit operator overloading.

Here are the most common operator traits you'll encounter:

Operator	Trait	Method Signature
`a + b`	`Add`	`fn add(self, rhs: Rhs) -> Output`

Operator	Trait	Method Signature
`a - b`	`Sub`	`fn sub(self, rhs: Rhs) -> Output`
`-a`	`Neg`	`fn neg(self) -> Output`
`a * b`	`Mul`	`fn mul(self, rhs: Rhs) -> Output`
`a / b`	`Div`	`fn div(self, rhs: Rhs) -> Output`

The `type Output = Vec3;` line in our implementation is an associated type—it tells Rust what type the operation returns. This flexibility allows interesting designs: for instance, you could make `Vec3 + f64` return a `Vec3`, but `Matrix * Vector` return a `Vector`.

Why `self` Instead of `&self`?

You might wonder why our `add` method takes `self` by value rather than by reference. Normally in Rust, we prefer references to avoid moving ownership. But remember: `Vec3` implements `Copy`. When we pass a `Copy` type by value, Rust automatically makes a copy—there's no ownership transfer to worry about.

For non-`Copy` types like `String`, you'd want to implement the operator for references too, so users don't lose ownership of their data:

```
impl Add for &Vec3 {
    type Output = Vec3;
    fn add(self, other: &Vec3) -> Vec3 {
        // Implementation would go here...
    }
}
```

But for our small, `Copy` vector type, the by-value version is simpler and just as efficient.

Rust Note: The Add Trait in Detail

Let's look at the full definition of the `Add` trait to understand how it works:

```
pub trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

There are three key pieces here. `Rhs` is a generic parameter representing the right-hand side type—it defaults to `Self`, which is why we can write `impl Add for Vec3` without specifying what we're adding. `Output` is an associated type that determines what the operation returns. And `self` is the left-hand side value.

This design enables heterogeneous operations. If you wanted to add a scalar to each component of a vector, you could implement `Add<f64>` for `Vec3`. The flexibility is there when you need it.

Practical Example: Position Updates

Vector addition is the heartbeat of game loops and physics simulations. Every frame, objects update their positions by adding their velocities. This simple operation, repeated sixty times per second, creates the illusion of motion.

```
fn main() {
    // Our player starts at the origin
    let mut position = Vec3::new(0.0, 0.0, 0.0);

    // They're moving diagonally across the screen
    let velocity = Vec3::new(1.0, 0.5, 0.0);

    // Simulate 5 frames of movement
    for frame in 0..5 {
        println!("Frame {}: position = {}", frame, position);
        position = position + velocity;
    }
}
```

Each iteration adds the velocity vector to the position, moving the player a little further:

```
Frame 0: position = (0, 0, 0)
Frame 1: position = (1, 0.5, 0)
Frame 2: position = (2, 1, 0)
Frame 3: position = (3, 1.5, 0)
Frame 4: position = (4, 2, 0)
```

This is the core of how every video game character moves. More sophisticated physics adds acceleration, friction, and collision detection, but they all build on this fundamental operation.

Practical Example: Finding Direction Between Points

Subtraction answers a question that comes up constantly in games and graphics: “Which way is that thing from here?” If you have two positions and want to know the direction and distance from one to the other, just subtract.

```
fn main() {
    let player = Vec3::new(2.0, 3.0, 0.0);
    let enemy = Vec3::new(8.0, 7.0, 0.0);

    // To find the direction FROM player TO enemy, subtract player from enemy
    let to_enemy = enemy - player;
    println!("Direction to enemy: {}", to_enemy);  // (6, 4, 0)

    // To find the direction FROM enemy TO player, subtract the other way
    let to_player = player - enemy;
    println!("Direction to player: {}", to_player);  // (-6, -4, 0)

    // These are exactly opposite—negating one gives the other
    assert_eq!(to_enemy, -to_player);
}
```

The vector `to_enemy` tells us we need to go 6 units in x and 4 units in y to get from the player to the enemy. This direction vector is essential for AI, pathfinding, and targeting systems.

Try It

These exercises will deepen your understanding and extend the functionality of your vector type.

1. **Implement `AddAssign` for `+=` syntax.** The `+=` operator is convenient for accumulating values. It modifies a variable in place rather than creating a new one:

   ```
   use std::ops::AddAssign;

   impl AddAssign for Vec3 {
       fn add_assign(&mut self, other: Vec3) {
           self.x += other.x;
           self.y += other.y;
           self.z += other.z;
       }
   }
   ```

   Now you can write `position += velocity;` instead of `position = position + velocity;`.

2. **Implement `SubAssign` for `-=` syntax.** Follow the same pattern as `AddAssign`.

3. **Write tests for the mathematical properties.** Good tests document expected behavior and catch regressions:

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_add_commutative() {
        let a = Vec3::new(1.0, 2.0, 3.0);
        let b = Vec3::new(4.0, 5.0, 6.0);
        assert_eq!(a + b, b + a);
    }

    #[test]
    fn test_add_associative() {
        let a = Vec3::new(1.0, 2.0, 3.0);
        let b = Vec3::new(4.0, 5.0, 6.0);
        let c = Vec3::new(7.0, 8.0, 9.0);
        assert_eq!((a + b) + c, a + (b + c));
    }

    #[test]
    fn test_add_identity() {
        let a = Vec3::new(1.0, 2.0, 3.0);
        assert_eq!(a + Vec3::zero(), a);
    }

    #[test]
    fn test_add_inverse() {
        let a = Vec3::new(1.0, 2.0, 3.0);
        assert_eq!(a + (-a), Vec3::zero());
    }
}
```

4. **Implement a midpoint function.** The midpoint between two positions is their average:

$$\vec{m} = \frac{\vec{a} + \vec{b}}{2}$$

You'll need scalar division from Recipe 1.3 to implement this, so bookmark it for later!

What's Next?

We can now combine vectors with addition, but what if we want to make a vector twice as long? Or half as long? Or point in the opposite direction? In Recipe 1.3, we'll implement scalar multiplication—the operation that lets us scale vectors to any size we need.

Scaling Vectors

Recipe 1.3: Scaling Vectors

The Goal

Sometimes you need a vector that points in the same direction but is twice as long, or half as long, or reversed. Scalar multiplication is the operation that makes this possible. By multiplying a vector by a number (a "scalar"), we can stretch it, shrink it, or flip it around while preserving its fundamental direction.

In this recipe, we'll implement scalar multiplication and division for our `Vec3` type. We'll also tackle a subtle Rust challenge: making both `v * 2.0` and `2.0 * v` work, just like they do in mathematics. By the end, you'll have the tools for smooth animations, physics simulations, and much more.

The Math

Scalar multiplication is wonderfully straightforward: multiply each component of the vector by the same number. If you want a vector twice as long, multiply every component by 2. If you want it pointing the opposite direction, multiply by -1. The operation preserves the vector's direction (or reverses it, if the scalar is negative) while changing its magnitude.

$$c \cdot \vec{v} = c \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} cx \\ cy \\ cz \end{pmatrix}$$

Here, $c \in \mathbb{R}$ is called a *scalar*—just a fancy word for a regular number. The term "scalar" comes from the idea that it "scales" the vector, making it longer or shorter.

Geometric Interpretation

The value of the scalar determines exactly how the vector changes. Think of it as a zoom factor: values greater than 1 zoom in (stretch), values between 0 and 1 zoom out (shrink), and negative values flip the picture before zooming.

Scalar c	**Effect on Vector**
$c > 1$	Stretches the vector, making it longer
$0 < c < 1$	Shrinks the vector, making it shorter
$c = 1$	No change at all
$c = 0$	Collapses to the zero vector
$c = -1$	Reverses direction, same length
$c < 0$	Reverses direction AND scales

Let's see a concrete example:

$$3 \cdot \begin{pmatrix} 2 \\ -1 \\ 4 \end{pmatrix} = \begin{pmatrix} 6 \\ -3 \\ 12 \end{pmatrix}$$

Each component is multiplied by 3: the x-component goes from 2 to 6, the y-component from -1 to -3, and the z-component from 4 to 12. The resulting vector points in exactly the same direction as the original, but is three times as long.

Mathematical Properties

Scalar multiplication plays nicely with the operations we've already defined. These properties might seem abstract, but they're what make vector math predictable and composable. When you chain operations together, these properties guarantee the result is what you'd expect.

Property	**Formula**	**What It Means**
Associative	$a(b\vec{v}) = (ab)\vec{v}$	Scaling by a then b equals scaling by ab
Distributive (scalar)	$(a + b)\vec{v} = a\vec{v} + b\vec{v}$	Can factor out the vector
Distributive (vector)	$c(\vec{u} + \vec{v}) = c\vec{u} + c\vec{v}$	Can factor out the scalar
Identity	$1 \cdot \vec{v} = \vec{v}$	Multiplying by 1 changes nothing
Zero	$0 \cdot \vec{v} = \vec{0}$	Multiplying by 0 kills the vector

Scalar Division

Once we have scalar multiplication, division is just multiplication in disguise. Dividing by c is the same as multiplying by $1/c$:

$$\frac{\vec{v}}{c} = \frac{1}{c} \cdot \vec{v} = \begin{pmatrix} x/c \\ y/c \\ z/c \end{pmatrix}$$

Just like with regular numbers, division by zero is undefined and will produce infinity or NaN in floating-point arithmetic. We won't add special handling for this—Rust's standard behavior matches what most graphics and physics code expects.

The Code

Add these trait implementations to `src/vec3.rs`. We need to implement `Mul` twice (for both operand orders) and `Div` once:

```rust
use std::ops::{Mul, Div};

// Vec3 * f64: Multiply a vector by a scalar on the right
impl Mul<f64> for Vec3 {
    type Output = Vec3;

    fn mul(self, scalar: f64) -> Vec3 {
        Vec3 {
            x: self.x * scalar,
            y: self.y * scalar,
            z: self.z * scalar,
        }
    }
}

// f64 * Vec3: Multiply a vector by a scalar on the left
// This lets us write "2.0 * vec" in addition to "vec * 2.0"
impl Mul<Vec3> for f64 {
    type Output = Vec3;

    fn mul(self, vec: Vec3) -> Vec3 {
        Vec3 {
            x: self * vec.x,
            y: self * vec.y,
            z: self * vec.z,
        }
    }
}

// Vec3 / f64: Divide a vector by a scalar
impl Div<f64> for Vec3 {
    type Output = Vec3;

    fn div(self, scalar: f64) -> Vec3 {
        Vec3 {
            x: self.x / scalar,
            y: self.y / scalar,
            z: self.z / scalar,
        }
    }
}
```

Now let's test all the variations in `main.rs`:

```rust
fn main() {
    let v = Vec3::new(2.0, 4.0, 6.0);

    // Scalar multiplication works in both orders
    let doubled = v * 2.0;
    let also_doubled = 2.0 * v;
    println!("v * 2 = {}", doubled);       // (4, 8, 12)
    println!("2 * v = {}", also_doubled);  // (4, 8, 12)

    // Scalar division
    let halved = v / 2.0;
    println!("v / 2 = {}", halved);  // (1, 2, 3)

    // Shrinking with a fractional scalar
    let shrunk = v * 0.5;
    println!("v * 0.5 = {}", shrunk);  // (1, 2, 3)

    // Reversing direction with a negative scalar
    let reversed = v * -1.0;
    println!("v * -1 = {}", reversed);  // (-2, -4, -6)

    // Chaining with addition works naturally
    let result = v * 2.0 + Vec3::unit_x() * 10.0;
    println!("v*2 + unit_x*10 = {}", result);  // (14, 8, 12)
}
```

The output shows that all our operations work as expected:

```
v * 2 = (4, 8, 12)
2 * v = (4, 8, 12)
v / 2 = (1, 2, 3)
v * 0.5 = (1, 2, 3)
v * -1 = (-2, -4, -6)
v*2 + unit_x*10 = (14, 8, 12)
```

Rust Note: Implementing Both `Vec3 * f64` and `f64 * Vec3`

In mathematics, scalar multiplication is commutative: $c\vec{v} = \vec{v}c$. Both orderings mean the same thing and should produce the same result. To support both syntaxes in Rust, we implement `Mul` twice with different type parameters.

```rust
// For "vec * scalar"
impl Mul<f64> for Vec3 { ... }

// For "scalar * vec"
impl Mul<Vec3> for f64 { ... }
```

Notice the subtle difference in these declarations. In the first, `Vec3` is `Self` and `f64` is the generic parameter `Rhs`. In the second, `f64` is `Self` and `Vec3` is `Rhs`. Without both

implementations, `2.0 * vec` would fail to compile even though `vec * 2.0` works perfectly.

This is a common pattern for mathematical types in Rust. You'll see it in libraries like `nalgebra` and `cgmath` as well.

Rust Note: Compound Assignment with `MulAssign` and `DivAssign`

Just like we added `AddAssign` for `+=`, we can add `MulAssign` and `DivAssign` for `*=` and `/=`. These operators modify a value in place, which can make code more concise and sometimes more efficient.

```
use std::ops::{MulAssign, DivAssign};

impl MulAssign<f64> for Vec3 {
    fn mul_assign(&mut self, scalar: f64) {
        self.x *= scalar;
        self.y *= scalar;
        self.z *= scalar;
    }
}

impl DivAssign<f64> for Vec3 {
    fn div_assign(&mut self, scalar: f64) {
        self.x /= scalar;
        self.y /= scalar;
        self.z /= scalar;
    }
}
```

With these in place, you can write more natural-looking code:

```
let mut v = Vec3::new(2.0, 4.0, 6.0);
v *= 3.0;  // v is now (6, 12, 18)
v /= 2.0;  // v is now (3, 6, 9)
```

Practical Example: Linear Interpolation (Lerp)

One of the most useful applications of scalar multiplication is **linear interpolation**, commonly called "lerp." Given two vectors and a parameter t between 0 and 1, lerp returns a vector that smoothly transitions from the first to the second. When $t = 0$, you get the first vector; when $t = 1$, you get the second; values in between give you points along the line connecting them.

$$\text{lerp}(\vec{a}, \vec{b}, t) = (1 - t)\vec{a} + t\vec{b}$$

This formula is the foundation of smooth animation. Instead of teleporting an object from point A to point B, you lerp between them over multiple frames, creating fluid motion.

```
impl Vec3 {
    /// Linear interpolation between self and other.
    /// When t=0, returns self. When t=1, returns other.
    /// Values of t between 0 and 1 give points along the line segment.
    pub fn lerp(self, other: Vec3, t: f64) -> Vec3 {
        self * (1.0 - t) + other * t
    }
}

fn main() {
    let start = Vec3::new(0.0, 0.0, 0.0);
    let end = Vec3::new(10.0, 20.0, 0.0);

    // Sample the interpolation at several points
    for i in 0..=4 {
        let t = i as f64 / 4.0;
        let pos = start.lerp(end, t);
        println!("t = {:.2}: {}", t, pos);
    }
}
```

The output shows a smooth progression from start to end:

```
t = 0.00: (0, 0, 0)
t = 0.25: (2.5, 5, 0)
t = 0.50: (5, 10, 0)
t = 0.75: (7.5, 15, 0)
t = 1.00: (10, 20, 0)
```

Lerp is everywhere in graphics: animating camera movements, fading colors, blending bone positions in skeletal animation, and much more.

Practical Example: Midpoint

Now we can implement the midpoint function that we teased in Recipe 1.2. The midpoint between two positions is simply their average—add them together and divide by 2:

$$\vec{m} = \frac{\vec{a} + \vec{b}}{2}$$

This is actually a special case of lerp where $t = 0.5$, but having a dedicated method makes the intent clearer.

```
impl Vec3 {
    /// Returns the midpoint between self and other.
    pub fn midpoint(self, other: Vec3) -> Vec3 {
        (self + other) / 2.0
    }
}

fn main() {
    let a = Vec3::new(0.0, 0.0, 0.0);
    let b = Vec3::new(10.0, 10.0, 10.0);

    println!("Midpoint: {}", a.midpoint(b));  // (5, 5, 5)
}
```

Practical Example: Velocity and Acceleration

Physics simulations use scalar multiplication constantly. The fundamental equations of motion involve multiplying velocity or acceleration by time. Here's a simple free-fall simulation that demonstrates both velocity and acceleration:

\[\vec{v}_{\text{new}} = \vec{v}_{\text{old}} + \vec{a} \cdot \Delta t \] \[\vec{p}_{\text{new}} = \vec{p}_{\text{old}} + \vec{v} \cdot \Delta t \]

These are the Euler integration equations, the simplest way to simulate physics. The acceleration (gravity) is multiplied by the time step to get a change in velocity. The velocity is multiplied by the time step to get a change in position.

```
fn main() {
    // Object starts 100 units up with zero velocity
    let mut position = Vec3::new(0.0, 100.0, 0.0);
    let mut velocity = Vec3::zero();

    // Gravity accelerates downward at 9.8 units per second squared
    let gravity = Vec3::new(0.0, -9.8, 0.0);

    // Simulate with a time step of 0.1 seconds
    let dt = 0.1;

    println!("Simulating free fall:");
    for step in 0..10 {
        println!("t={:.1}s: pos={}, vel={}",
                 step as f64 * dt, position, velocity);

        // Update velocity: v = v + a*dt
        velocity = velocity + gravity * dt;

        // Update position: p = p + v*dt
        position = position + velocity * dt;
    }
}
```

Watch the object accelerate as it falls:

```
Simulating free fall:
t=0.0s: pos=(0, 100, 0), vel=(0, 0, 0)
t=0.1s: pos=(0, 99.902, 0), vel=(0, -0.98, 0)
t=0.2s: pos=(0, 99.706, 0), vel=(0, -1.96, 0)
t=0.3s: pos=(0, 99.412, 0), vel=(0, -2.94, 0)
...
```

The position decreases slowly at first (small velocity), then faster and faster as gravity accelerates the object. This is the foundation of every physics engine.

Try It

These exercises will deepen your understanding of scalar multiplication and its applications.

1. **Weighted average.** Given three points with weights, compute the weighted center. This is useful for finding centroids and for barycentric coordinates in triangles.

$$\vec{c} = \frac{w_1\vec{p}_1 + w_2\vec{p}_2 + w_3\vec{p}_3}{w_1 + w_2 + w_3}$$

2. **Bezier curves.** A quadratic Bezier curve is defined by three control points. As t goes from 0 to 1, the curve traces a smooth path from $\vec{p}_0$ to $\vec{p}_2$, pulled toward $\vec{p}_1$:

$$\vec{B}(t) = (1-t)^2\vec{p}_0 + 2(1-t)t\vec{p}_1 + t^2\vec{p}_2$$

   Implement this function and print several points along the curve. Bezier curves are used everywhere from font rendering to animation paths.

3. **Property tests.** Verify that our implementation satisfies the mathematical properties:

```rust
#[test]
fn test_scalar_mul_associative() {
    let v = Vec3::new(1.0, 2.0, 3.0);
    let a = 2.0;
    let b = 3.0;
    // a(bv) = (ab)v
    assert_eq!(a * (b * v), (a * b) * v);
}

#[test]
fn test_scalar_mul_distributive_scalar() {
    let v = Vec3::new(1.0, 2.0, 3.0);
    let a = 2.0;
    let b = 3.0;
    // (a + b)v = av + bv
    assert_eq!((a + b) * v, a * v + b * v);
}

#[test]
fn test_scalar_mul_distributive_vector() {
    let u = Vec3::new(1.0, 2.0, 3.0);
    let v = Vec3::new(4.0, 5.0, 6.0);
    let c = 2.0;
    // c(u + v) = cu + cv
    assert_eq!(c * (u + v), c * u + c * v);
}
```

What's Next?

We can now add vectors and scale them, giving us control over their magnitude and direction. But we can't yet measure the *relationship* between two vectors. Are they pointing the same way? Are they perpendicular? What's the angle between them?

In Recipe 1.4, we'll implement the **dot product**—an operation that answers all these questions and more. The dot product is the key to lighting calculations in graphics, similarity measures in machine learning, and projections in physics.

The Dot Product

Recipe 1.4: The Dot Product

The Goal

So far, our vector operations have been about combining and transforming individual vectors. But what if we want to know how two vectors *relate* to each other? Are they pointing in the same direction? Are they perpendicular? What's the angle between them? The dot product answers all these questions and more.

The dot product is arguably the most important operation in computational geometry. It's the foundation of lighting in 3D graphics, similarity measures in machine learning, and projections in physics. Despite its power, the formula is remarkably simple—just multiply corresponding components and add them up.

In this recipe, we'll implement the dot product and explore its many applications. By the end, you'll understand why this single operation appears in nearly every graphics and physics codebase.

The Math

The **dot product** (also called *inner product* or *scalar product*) takes two vectors and returns a single number—a scalar, not another vector. This might seem strange at first: how can combining two arrows give us just a number? But that number encodes profound geometric information about the relationship between those arrows.

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z = \sum_{i=1}^{n} a_i b_i$$

Example:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = (1)(4) + (2)(5) + (3)(6) = 4 + 10 + 18 = 32$$

Geometric Interpretation

The dot product has a beautiful geometric meaning:

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos \theta$$

Where θ is the angle between the vectors.

This leads to a crucial insight about what the sign tells us:

Condition	Angle θ	Interpretation
$\vec{a} \cdot \vec{b} > 0$	$0° \leq \theta < 90°$	Vectors point in similar directions
$\vec{a} \cdot \vec{b} = 0$	$\theta = 90°$	Vectors are **perpendicular** (orthogonal)
$\vec{a} \cdot \vec{b} < 0$	$90° < \theta \leq 180°$	Vectors point in opposite directions

Properties

Property	Formula
Commutative	$\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a}$
Distributive	$\vec{a} \cdot (\vec{b} + \vec{c}) = \vec{a} \cdot \vec{b} + \vec{a} \cdot \vec{c}$
Scalar multiplication	$(c\vec{a}) \cdot \vec{b} = c(\vec{a} \cdot \vec{b})$
Self-dot	$\vec{a} \cdot \vec{a} = \|\vec{a}\|^2$

The last property is key: **the dot product of a vector with itself gives the square of its length**.

Finding Angles

Rearranging the geometric formula:

$$\cos \theta = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}$$

$$\theta = \arccos\left(\frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|} \right)$$

The Code

Add to `src/vec3.rs`:

```rust
impl Vec3 {
    /// Computes the dot product of two vectors.
    ///
    /// The dot product is a scalar that measures how much two vectors
    /// point in the same direction.
    pub fn dot(self, other: Vec3) -> f64 {
        self.x * other.x + self.y * other.y + self.z * other.z
    }

    /// Returns the squared length of the vector.
    ///
    /// This is faster than `length()` when you only need to compare lengths,
    /// since it avoids the square root.
    pub fn length_squared(self) -> f64 {
        self.dot(self)
    }

    /// Returns the length (magnitude) of the vector.
    pub fn length(self) -> f64 {
        self.length_squared().sqrt()
    }

    /// Returns the angle between two vectors in radians.
    pub fn angle_between(self, other: Vec3) -> f64 {
        let dot = self.dot(other);
        let lengths = self.length() * other.length();

        if lengths == 0.0 {
            return 0.0;  // Avoid division by zero
        }

        // Clamp to [-1, 1] to handle floating-point errors
        let cos_angle = (dot / lengths).clamp(-1.0, 1.0);
        cos_angle.acos()
    }

    /// Checks if two vectors are perpendicular (orthogonal).
    ///
    /// Uses a small epsilon for floating-point comparison.
    pub fn is_perpendicular(self, other: Vec3, epsilon: f64) -> bool {
        self.dot(other).abs() < epsilon
    }
}
```

Use it in `main.rs`:

```rust
fn main() {
    let a = Vec3::new(1.0, 0.0, 0.0);
    let b = Vec3::new(0.0, 1.0, 0.0);
    let c = Vec3::new(1.0, 1.0, 0.0);

    // Basic dot product
    println!("a · a = {}", a.dot(a));  // 1 (parallel to itself)
    println!("a · b = {}", a.dot(b));  // 0 (perpendicular)
    println!("a · c = {}", a.dot(c));  // 1

    // Checking perpendicularity
    println!("a ⊥ b? {}", a.is_perpendicular(b, 1e-10));  // true
    println!("a ⊥ c? {}", a.is_perpendicular(c, 1e-10));  // false

    // Finding angles (converting radians to degrees)
    let angle_ab = a.angle_between(b) * 180.0 / std::f64::consts::PI;
    let angle_ac = a.angle_between(c) * 180.0 / std::f64::consts::PI;
    println!("Angle between a and b: {:.1}°", angle_ab);  // 90.0°
    println!("Angle between a and c: {:.1}°", angle_ac);  // 45.0°

    // Positive vs negative dot products
    let forward = Vec3::new(1.0, 0.0, 0.0);
    let backward = Vec3::new(-1.0, 0.0, 0.0);
    let sideways = Vec3::new(0.0, 1.0, 0.0);

    println!("\nDirection checks:");
    println!("forward · forward = {}", forward.dot(forward));   // 1 (same
direction)
    println!("forward · backward = {}", forward.dot(backward)); // -1
(opposite)
    println!("forward · sideways = {}", forward.dot(sideways)); // 0
(perpendicular)
}
```

Output:

```
a · a = 1
a · b = 0
a · c = 1
a ⊥ b? true
a ⊥ c? false
Angle between a and b: 90.0°
Angle between a and c: 45.0°

Direction checks:
forward · forward = 1
forward · backward = -1
forward · sideways = 0
```

Rust Note: Method vs Function Syntax

We implemented `dot` as a method:

```
let result = a.dot(b);
```

Alternatively, you could implement it as an associated function:

```
impl Vec3 {
    pub fn dot(a: Vec3, b: Vec3) -> f64 { ... }
}

// Called as:
let result = Vec3::dot(a, b);
```

Or even operator overload (though `*` is typically used for element-wise or cross product in other libraries):

```
impl Mul<Vec3> for Vec3 {
    type Output = f64;
    fn mul(self, other: Vec3) -> f64 {
        self.dot(other)
    }
}

// Called as:
let result = a * b;  // Returns f64, not Vec3!
```

The method syntax `a.dot(b)` is clearest and most common.

Practical Example: Light and Surfaces

In 3D graphics, the dot product determines how bright a surface appears:

```rust
fn main() {
    // Surface normal (pointing "out" from the surface)
    let normal = Vec3::new(0.0, 1.0, 0.0);  // Pointing up

    // Light direction (pointing toward the light source)
    let light_dir = Vec3::new(0.0, 1.0, 0.0);  // Light from above

    // Diffuse lighting: intensity = max(0, normal · light)
    let intensity = normal.dot(light_dir).max(0.0);
    println!("Light from above: intensity = {}", intensity);  // 1.0

    // Light from the side
    let side_light = Vec3::new(1.0, 0.0, 0.0);
    let intensity2 = normal.dot(side_light).max(0.0);
    println!("Light from side: intensity = {}", intensity2);  // 0.0

    // Light at 45 degrees
    let angled_light = Vec3::new(0.0, 1.0, 1.0);
    let angled_normalized = angled_light / angled_light.length();
    let intensity3 = normal.dot(angled_normalized).max(0.0);
    println!("Light at 45°: intensity = {:.3}", intensity3);  // ~0.707
}
```

Practical Example: Is Enemy in Front of Player?

Use the dot product to check if something is in front of you:

```rust
fn main() {
    let player_pos = Vec3::new(0.0, 0.0, 0.0);
    let player_facing = Vec3::new(1.0, 0.0, 0.0);  // Looking along +X

    let enemy1 = Vec3::new(5.0, 2.0, 0.0);   // In front and to the side
    let enemy2 = Vec3::new(-3.0, 1.0, 0.0);  // Behind

    // Direction to each enemy
    let to_enemy1 = enemy1 - player_pos;
    let to_enemy2 = enemy2 - player_pos;

    // Positive dot = in front, negative = behind
    let dot1 = player_facing.dot(to_enemy1);
    let dot2 = player_facing.dot(to_enemy2);

    println!("Enemy 1 dot product: {} ({})",
             dot1, if dot1 > 0.0 { "IN FRONT" } else { "BEHIND" });
    println!("Enemy 2 dot product: {} ({})",
             dot2, if dot2 > 0.0 { "IN FRONT" } else { "BEHIND" });
}
```

Output:

```
Enemy 1 dot product: 5 (IN FRONT)
Enemy 2 dot product: -3 (BEHIND)
```

Practical Example: Field of View Check

Combine dot product with angle to check if something is within a cone of vision:

```
fn is_in_fov(viewer_pos: Vec3, viewer_dir: Vec3, target: Vec3, fov_degrees:
f64) -> bool {
    let to_target = target - viewer_pos;
    let angle = viewer_dir.angle_between(to_target);
    let half_fov = (fov_degrees / 2.0) * std::f64::consts::PI / 180.0;
    angle <= half_fov
}

fn main() {
    let player_pos = Vec3::zero();
    let player_dir = Vec3::unit_x();  // Looking along +X

    let targets = vec![
        Vec3::new(10.0, 0.0, 0.0),   // Directly ahead
        Vec3::new(10.0, 5.0, 0.0),   // Ahead and up
        Vec3::new(10.0, 12.0, 0.0),  // Way up (out of FOV)
        Vec3::new(-5.0, 0.0, 0.0),   // Behind
    ];

    let fov = 90.0;  // 90-degree field of view

    for (i, target) in targets.iter().enumerate() {
        let visible = is_in_fov(player_pos, player_dir, *target, fov);
        println!("Target {} at {}: {}",
                 i, target, if visible { "VISIBLE" } else { "not visible" });
    }
}
```

Try It

1. **Scalar projection:** The scalar projection of $\vec{b}$ onto $\vec{a}$ tells you how far $\vec{b}$ extends in the direction of $\vec{a}$:

$$\text{comp}_{\vec{a}}\vec{b} = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}|}$$

```
pub fn scalar_projection(self, onto: Vec3) -> f64 {
    self.dot(onto) / onto.length()
}
```

2. **Check if vectors are parallel:** Two vectors are parallel if the angle between them is 0° or 180°:

```
pub fn is_parallel(self, other: Vec3, epsilon: f64) -> bool {
    let cos_angle = self.dot(other) / (self.length() * other.length());
    (cos_angle.abs() - 1.0).abs() < epsilon
}
```

3. **Tests:**

```
#[test]
fn test_dot_commutative() {
    let a = Vec3::new(1.0, 2.0, 3.0);
    let b = Vec3::new(4.0, 5.0, 6.0);
    assert_eq!(a.dot(b), b.dot(a));
}

#[test]
fn test_perpendicular_vectors() {
    let a = Vec3::new(1.0, 0.0, 0.0);
    let b = Vec3::new(0.0, 1.0, 0.0);
    assert!(a.is_perpendicular(b, 1e-10));
}

#[test]
fn test_self_dot_is_length_squared() {
    let v = Vec3::new(3.0, 4.0, 0.0);
    assert_eq!(v.dot(v), 25.0);  // 3² + 4² = 25
    assert_eq!(v.length(), 5.0);
}
```

4. **Cosine similarity:** Used in machine learning to compare vectors (documents, word embeddings, etc.):

$$\text{similarity}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}$$

This returns a value between -1 (opposite) and 1 (identical direction).

What's Next?

The dot product gave us `length_squared()`. In Recipe 1.5, we'll explore vector length more fully and learn about **normalization**—converting any vector to a unit vector.

Vector Length and Normalization

Recipe 1.5: Vector Length and Normalization

The Goal

When working with vectors, two questions come up constantly: “How long is this vector?” and “Which way does it point?” These questions sound simple, but answering them precisely requires two key operations: computing length (magnitude) and normalizing (converting to a unit vector).

Length tells us the magnitude of a displacement, velocity, or force. Normalization strips away the magnitude, leaving only pure direction. Together, they let us decompose any vector into “how much” and “which way”—a separation that’s essential for physics, graphics, and machine learning.

In this recipe, we’ll implement length, distance, and normalization. We’ll also learn why `length_squared()` is often preferable to `length()`, a performance trick you’ll see in every optimized graphics codebase.

The Math

Vector Length (Magnitude)

The **length** or **magnitude** of a vector measures how far the tip of the arrow is from the origin. We write it with double vertical bars: $|\vec{v}|$. The formula comes directly from the Pythagorean theorem, extended to three dimensions:

$$|\vec{v}| = \sqrt{x^2 + y^2 + z^2} = \sqrt{\vec{v} \cdot \vec{v}}$$

This is the Euclidean distance from the origin to the point (x, y, z).

Example:

$$\left|\begin{pmatrix} 3 \\ 4 \\ 0 \end{pmatrix}\right| = \sqrt{3^2 + 4^2 + 0^2} = \sqrt{9 + 16} = \sqrt{25} = 5$$

The famous 3-4-5 right triangle!

Distance Between Points

The distance between two points $\vec{a}$ and $\vec{b}$ is the length of their difference:

$$d(\vec{a}, \vec{b}) = |\vec{b} - \vec{a}| = \sqrt{(b_x - a_x)^2 + (b_y - a_y)^2 + (b_z - a_z)^2}$$

Unit Vectors

A **unit vector** has length exactly 1:

$$|\hat{v}| = 1$$

We often write unit vectors with a hat ($\hat{v}$) instead of an arrow.

Normalization

Normalizing a vector means scaling it to have length 1 while preserving its direction:

$$\hat{v} = \frac{\vec{v}}{|\vec{v}|}$$

This divides each component by the length:

$$\hat{v} = \frac{1}{|\vec{v}|}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x/|\vec{v}| \\ y/|\vec{v}| \\ z/|\vec{v}| \end{pmatrix}$$

Example: Normalize $\vec{v} = \begin{pmatrix} 3 \\ 4 \\ 0 \end{pmatrix}$:

$$|\vec{v}| = 5$$

$$\hat{v} = \frac{1}{5}\begin{pmatrix} 3 \\ 4 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.6 \\ 0.8 \\ 0 \end{pmatrix}$$

Verify: $\sqrt{0.6^2 + 0.8^2} = \sqrt{0.36 + 0.64} = \sqrt{1} = 1$ ✓

Why Use Unit Vectors?

Unit vectors represent **pure direction** without magnitude. They're essential because:

1. **Directions**: "Which way?" is separate from "how far?"

2. **Lighting**: Surface normals should be unit vectors
3. **Physics**: Force direction vs force magnitude
4. **Simplification**: Many formulas simplify when vectors are normalized

The Code

We already added `length()` and `length_squared()` in Recipe 1.4. Now add:

```
impl Vec3 {
    /// Returns a unit vector pointing in the same direction.
    ///
    /// Returns the zero vector if the input has zero length.
    pub fn normalize(self) -> Vec3 {
        let len = self.length();
        if len == 0.0 {
            Vec3::zero()
        } else {
            self / len
        }
    }

    /// Returns a unit vector, or None if the vector has zero length.
    ///
    /// Use this when you need to handle the zero-vector case explicitly.
    pub fn try_normalize(self) -> Option<Vec3> {
        let len = self.length();
        if len == 0.0 {
            None
        } else {
            Some(self / len)
        }
    }

    /// Checks if this is approximately a unit vector.
    pub fn is_normalized(self, epsilon: f64) -> bool {
        (self.length_squared() - 1.0).abs() < epsilon
    }

    /// Returns the distance to another vector (treating both as points).
    pub fn distance(self, other: Vec3) -> f64 {
        (other - self).length()
    }

    /// Returns the squared distance to another vector.
    ///
    /// Faster than `distance()` when you only need to compare distances.
    pub fn distance_squared(self, other: Vec3) -> f64 {
        (other - self).length_squared()
    }
}
```

Use it in `main.rs`:

```rust
fn main() {
    // Basic length
    let v = Vec3::new(3.0, 4.0, 0.0);
    println!("v = {}", v);
    println!("length = {}", v.length());  // 5
    println!("length² = {}", v.length_squared());  // 25

    // Normalization
    let unit = v.normalize();
    println!("normalized = {}", unit);  // (0.6, 0.8, 0)
    println!("normalized length = {}", unit.length());  // 1

    // Check if normalized
    println!("v is unit vector? {}", v.is_normalized(1e-10));      // false
    println!("unit is unit vector? {}", unit.is_normalized(1e-10)); // true

    // Standard unit vectors
    println!("\nStandard unit vectors:");
    println!("unit_x: {} (length {})", Vec3::unit_x(),
Vec3::unit_x().length());
    println!("unit_y: {} (length {})", Vec3::unit_y(),
Vec3::unit_y().length());
    println!("unit_z: {} (length {})", Vec3::unit_z(),
Vec3::unit_z().length());

    // Distance between points
    let a = Vec3::new(1.0, 2.0, 3.0);
    let b = Vec3::new(4.0, 6.0, 3.0);
    println!("\nDistance from {} to {} = {}", a, b, a.distance(b));  // 5

    // Handling zero vectors
    let zero = Vec3::zero();
    println!("\nZero vector normalized: {}", zero.normalize());  // (0, 0, 0)
    println!("Zero vector try_normalize: {:?}", zero.try_normalize());  //
None
}
```

Output:

```
v = (3, 4, 0)
length = 5
length² = 25
normalized = (0.6, 0.8, 0)
normalized length = 1
v is unit vector? false
unit is unit vector? true

Standard unit vectors:
unit_x: (1, 0, 0) (length 1)
unit_y: (0, 1, 0) (length 1)
unit_z: (0, 0, 1) (length 1)

Distance from (1, 2, 3) to (4, 6, 3) = 5

Zero vector normalized: (0, 0, 0)
Zero vector try_normalize: None
```

Rust Note: Option for Fallible Operations

The `try_normalize` function returns `Option<Vec3>`:

```rust
pub fn try_normalize(self) -> Option<Vec3> {
    let len = self.length();
    if len == 0.0 {
        None
    } else {
        Some(self / len)
    }
}
```

This forces callers to handle the zero-vector case:

```rust
let v = Vec3::new(0.0, 0.0, 0.0);

// Using match
match v.try_normalize() {
    Some(unit) => println!("Unit vector: {}", unit),
    None => println!("Cannot normalize zero vector"),
}

// Using if let
if let Some(unit) = v.try_normalize() {
    // Use unit...
}

// Using unwrap_or
let unit = v.try_normalize().unwrap_or(Vec3::unit_x());
```

Rust Note: Why `length_squared()`?

Computing length requires a square root, which is relatively slow:

```
// Slow: computes sqrt
let len = v.length();

// Fast: no sqrt
let len_sq = v.length_squared();
```

When comparing distances, you often don't need the actual values—you just need to know which is larger. Since $a < b \Leftrightarrow a^2 < b^2$ (for positive values), you can compare squared lengths:

```
// Instead of:
if a.distance(target) < b.distance(target) { ... }

// Use:
if a.distance_squared(target) < b.distance_squared(target) { ... }
```

Practical Example: Find Nearest Point

```
fn find_nearest(origin: Vec3, points: &[Vec3]) -> Option<Vec3> {
    points
        .iter()
        .min_by(|a, b| {
            let dist_a = origin.distance_squared(**a);
            let dist_b = origin.distance_squared(**b);
            dist_a.partial_cmp(&dist_b).unwrap()
        })
        .copied()
}

fn main() {
    let player = Vec3::new(0.0, 0.0, 0.0);
    let enemies = vec![
        Vec3::new(10.0, 0.0, 0.0),
        Vec3::new(3.0, 4.0, 0.0),  // Distance 5
        Vec3::new(8.0, 6.0, 0.0),
    ];

    if let Some(nearest) = find_nearest(player, &enemies) {
        println!("Nearest enemy at: {} (distance: {})",
                 nearest, player.distance(nearest));
    }
}
```

Output:

```
Nearest enemy at: (3, 4, 0) (distance: 5)
```

Practical Example: Direction to Target

```
fn main() {
    let player = Vec3::new(2.0, 3.0, 0.0);
    let target = Vec3::new(8.0, 7.0, 0.0);

    // Vector pointing from player to target
    let to_target = target - player;  // (6, 4, 0)

    // Distance to target
    let distance = to_target.length();  // ~7.21

    // Direction to target (unit vector)
    let direction = to_target.normalize();

    println!("Player at: {}", player);
    println!("Target at: {}", target);
    println!("Distance: {:.2}", distance);
    println!("Direction: ({:.3}, {:.3}, {:.3})",
             direction.x, direction.y, direction.z);

    // Move player 1 unit toward target
    let new_pos = player + direction * 1.0;
    println!("After moving 1 unit toward target: ({:.3}, {:.3}, {:.3})",
             new_pos.x, new_pos.y, new_pos.z);
}
```

Output:

```
Player at: (2, 3, 0)
Target at: (8, 7, 0)
Distance: 7.21
Direction: (0.832, 0.555, 0.000)
After moving 1 unit toward target: (2.832, 3.555, 0.000)
```

Practical Example: Sphere Collision Detection

Two spheres collide when the distance between their centers is less than the sum of their radii:

```rust
struct Sphere {
    center: Vec3,
    radius: f64,
}

impl Sphere {
    fn intersects(&self, other: &Sphere) -> bool {
        let distance_sq = self.center.distance_squared(other.center);
        let radii_sum = self.radius + other.radius;
        distance_sq < radii_sum * radii_sum
    }
}

fn main() {
    let s1 = Sphere { center: Vec3::new(0.0, 0.0, 0.0), radius: 2.0 };
    let s2 = Sphere { center: Vec3::new(3.0, 0.0, 0.0), radius: 1.5 };
    let s3 = Sphere { center: Vec3::new(10.0, 0.0, 0.0), radius: 1.0 };

    println!("s1 intersects s2? {}", s1.intersects(&s2));  // true (distance
3 < 3.5)
    println!("s1 intersects s3? {}", s1.intersects(&s3));  // false (distance
10 > 3)
}
```

Try It

1. **Set length to specific value:** Scale a vector to have a desired length:

    ```rust
    pub fn with_length(self, new_length: f64) -> Vec3 {
        self.normalize() * new_length
    }
    ```

2. **Clamp length:** Limit a vector's length to a maximum:

    ```rust
    pub fn clamp_length(self, max_length: f64) -> Vec3 {
        let len_sq = self.length_squared();
        if len_sq > max_length * max_length {
            self.normalize() * max_length
        } else {
            self
        }
    }
    ```

3. **Move toward target:** Move a point toward a target by a fixed distance:

```
    pub fn move_toward(self, target: Vec3, max_distance: f64) -> Vec3 {
        let to_target = target - self;
        let distance = to_target.length();
        if distance <= max_distance {
            target
        } else {
            self + to_target.normalize() * max_distance
        }
    }
```

4. **Tests:**

```
    #[test]
    fn test_unit_vector_length() {
        let v = Vec3::new(3.0, 4.0, 0.0);
        let unit = v.normalize();
        assert!((unit.length() - 1.0).abs() < 1e-10);
    }

    #[test]
    fn test_distance_symmetric() {
        let a = Vec3::new(1.0, 2.0, 3.0);
        let b = Vec3::new(4.0, 5.0, 6.0);
        assert_eq!(a.distance(b), b.distance(a));
    }

    #[test]
    fn test_3_4_5_triangle() {
        let v = Vec3::new(3.0, 4.0, 0.0);
        assert_eq!(v.length(), 5.0);
    }
```

What's Next?

With dot products, we can measure the angle between vectors. But what if we need a vector *perpendicular* to two others? In Recipe 1.6, we'll implement the **cross product**—a uniquely 3D operation that's essential for graphics and physics.

The Cross Product

Recipe 1.6: The Cross Product

The Goal

The dot product tells us about alignment—how much two vectors point in the same direction. But sometimes we need the opposite: a vector that's perpendicular to two others. Given two vectors lying in a plane, can we find a third vector that sticks straight out of that plane? This is exactly what the cross product does.

The cross product is uniquely three-dimensional—it doesn't exist in 2D, and it generalizes differently in higher dimensions. But in 3D, it's indispensable. Every surface normal in a 3D mesh is computed using cross products. Every physics simulation that involves rotation uses cross products for torque and angular momentum. If you've ever wondered how games know which direction a surface faces, the cross product is the answer.

In this final recipe of Chapter 1, we'll implement the cross product and explore its applications in graphics and physics. This completes our vector toolkit, giving us everything we need to tackle matrices in Chapter 2.

The Math

The **cross product** (or *vector product*) takes two 3D vectors and produces a third vector that is perpendicular to both inputs. Unlike the dot product, which returns a scalar, the cross product returns another vector. This vector points in the direction determined by the "right-hand rule," and its length equals the area of the parallelogram formed by the input vectors.

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix}$$

You can remember this using the **determinant form**:

$$\vec{a} \times \vec{b} = \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

Where $\hat{i}, \hat{j}, \hat{k}$ are the standard unit vectors along x, y, z.

Example:

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \times \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} (0)(0) - (0)(1) \\ (0)(0) - (1)(0) \\ (1)(1) - (0)(0) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

The cross product of the x and y unit vectors gives the z unit vector!

Geometric Properties

1. The result is perpendicular to both inputs:

$$(\vec{a} \times \vec{b}) \perp \vec{a} \quad \text{and} \quad (\vec{a} \times \vec{b}) \perp \vec{b}$$

This means: $\vec{a} \cdot (\vec{a} \times \vec{b}) = 0$ and $\vec{b} \cdot (\vec{a} \times \vec{b}) = 0$

2. The magnitude equals the parallelogram area:

$$|\vec{a} \times \vec{b}| = |\vec{a}||\vec{b}| \sin \theta$$

Where θ is the angle between the vectors. This is the area of the parallelogram formed by $\vec{a}$ and $\vec{b}$.

3. The right-hand rule determines direction:

Point your fingers along $\vec{a}$, curl them toward $\vec{b}$, and your thumb points in the direction of $\vec{a} \times \vec{b}$.

Properties

Property	Formula	Notes
Anti-commutative	$\vec{a} \times \vec{b} = -(\vec{b} \times \vec{a})$	Order matters!
Not associative	$\vec{a} \times (\vec{b} \times \vec{c}) \neq (\vec{a} \times \vec{b}) \times \vec{c}$	Grouping matters!
Distributive	$\vec{a} \times (\vec{b} + \vec{c}) = \vec{a} \times \vec{b} + \vec{a} \times \vec{c}$	
Scalar multiplication	$(c\vec{a}) \times \vec{b} = c(\vec{a} \times \vec{b})$	
Self-cross	$\vec{a} \times \vec{a} = \vec{0}$	A vector is parallel to itself

When the Cross Product is Zero

$\vec{a} \times \vec{b} = \vec{0}$ when:

- Either vector is zero
- The vectors are **parallel** (or anti-parallel)

Since $|\vec{a} \times \vec{b}| = |\vec{a}||\vec{b}| \sin\theta$, and $\sin(0°) = \sin(180°) = 0$.

Standard Unit Vector Cross Products

$$\hat{i} \times \hat{j} = \hat{k}, \quad \hat{j} \times \hat{k} = \hat{i}, \quad \hat{k} \times \hat{i} = \hat{j}$$

$$\hat{j} \times \hat{i} = -\hat{k}, \quad \hat{k} \times \hat{j} = -\hat{i}, \quad \hat{i} \times \hat{k} = -\hat{j}$$

The Code

Add to `src/vec3.rs`:

```rust
impl Vec3 {
    /// Computes the cross product of two vectors.
    ///
    /// The result is perpendicular to both input vectors.
    /// Only meaningful in 3D.
    pub fn cross(self, other: Vec3) -> Vec3 {
        Vec3 {
            x: self.y * other.z - self.z * other.y,
            y: self.z * other.x - self.x * other.z,
            z: self.x * other.y - self.y * other.x,
        }
    }

    /// Returns the area of the parallelogram formed by two vectors.
    pub fn parallelogram_area(self, other: Vec3) -> f64 {
        self.cross(other).length()
    }

    /// Returns the area of the triangle formed by two vectors.
    pub fn triangle_area(self, other: Vec3) -> f64 {
        self.parallelogram_area(other) / 2.0
    }

    /// Checks if two vectors are parallel (or anti-parallel).
    pub fn is_parallel(self, other: Vec3, epsilon: f64) -> bool {
        self.cross(other).length_squared() < epsilon * epsilon
    }
}
```

Use it in `main.rs`:

```rust
fn main() {
    let i = Vec3::unit_x();
    let j = Vec3::unit_y();
    let k = Vec3::unit_z();

    // Standard unit vector cross products
    println!("i × j = {}", i.cross(j));  // (0, 0, 1) = k
    println!("j × k = {}", j.cross(k));  // (1, 0, 0) = i
    println!("k × i = {}", k.cross(i));  // (0, 1, 0) = j

    // Anti-commutative property
    println!("\nAnti-commutative:");
    println!("j × i = {}", j.cross(i));  // (0, 0, -1) = -k
    println!("i × j = -{} ? {}", i.cross(j), j.cross(i) == -(i.cross(j)));

    // Result is perpendicular to both inputs
    let a = Vec3::new(1.0, 2.0, 3.0);
    let b = Vec3::new(4.0, 5.0, 6.0);
    let c = a.cross(b);

    println!("\nPerpendicularity check:");
    println!("a = {}", a);
    println!("b = {}", b);
    println!("a × b = {}", c);
    println!("a · (a × b) = {}", a.dot(c));  // Should be 0
    println!("b · (a × b) = {}", b.dot(c));  // Should be 0

    // Self-cross is zero
    println!("\nSelf-cross:");
    println!("a × a = {}", a.cross(a));  // (0, 0, 0)

    // Parallel vectors have zero cross product
    let parallel = a * 2.0;
    println!("\nParallel vectors:");
    println!("a × (2*a) = {}", a.cross(parallel));  // (0, 0, 0)
    println!("a is parallel to 2*a? {}", a.is_parallel(parallel, 1e-10));
}
```

Output:

```
i × j = (0, 0, 1)
j × k = (1, 0, 0)
k × i = (0, 1, 0)

Anti-commutative:
j × i = (0, 0, -1)
i × j = -(0, 0, 1) ? true

Perpendicularity check:
a = (1, 2, 3)
b = (4, 5, 6)
a × b = (-3, 6, -3)
a · (a × b) = 0
b · (a × b) = 0

Self-cross:
a × a = (0, 0, 0)

Parallel vectors:
a × (2*a) = (0, 0, 0)
a is parallel to 2*a? true
```

Practical Example: Surface Normals

In 3D graphics, every surface has a **normal vector** that points perpendicular to the surface. For a triangle defined by vertices $\vec{p}_0, \vec{p}_1, \vec{p}_2$, the normal is computed using the cross product:

```
fn triangle_normal(p0: Vec3, p1: Vec3, p2: Vec3) -> Vec3 {
    let edge1 = p1 - p0;
    let edge2 = p2 - p0;
    edge1.cross(edge2).normalize()
}

fn main() {
    // A triangle in the XY plane
    let p0 = Vec3::new(0.0, 0.0, 0.0);
    let p1 = Vec3::new(1.0, 0.0, 0.0);
    let p2 = Vec3::new(0.0, 1.0, 0.0);

    let normal = triangle_normal(p0, p1, p2);
    println!("Triangle normal: {}", normal);  // (0, 0, 1) - pointing up

    // A tilted triangle
    let q0 = Vec3::new(0.0, 0.0, 0.0);
    let q1 = Vec3::new(1.0, 0.0, 0.0);
    let q2 = Vec3::new(0.0, 1.0, 1.0);

    let normal2 = triangle_normal(q0, q1, q2);
    println!("Tilted triangle normal: ({:.3}, {:.3}, {:.3})",
             normal2.x, normal2.y, normal2.z);
}
```

Practical Example: Computing Triangle Area

The cross product magnitude gives area:

```
fn triangle_area_from_vertices(p0: Vec3, p1: Vec3, p2: Vec3) -> f64 {
    let edge1 = p1 - p0;
    let edge2 = p2 - p0;
    edge1.triangle_area(edge2)
}

fn main() {
    // Right triangle with legs 3 and 4
    let p0 = Vec3::new(0.0, 0.0, 0.0);
    let p1 = Vec3::new(3.0, 0.0, 0.0);
    let p2 = Vec3::new(0.0, 4.0, 0.0);

    let area = triangle_area_from_vertices(p0, p1, p2);
    println!("Triangle area: {}", area);  // 6.0 (= 3*4/2)
}
```

Practical Example: Look-At Rotation

Games often need to make an object “look at” a target. The cross product helps build a rotation frame:

```rust
/// Computes an orthonormal basis (right, up, forward) for looking at a
target.
fn look_at_basis(eye: Vec3, target: Vec3, world_up: Vec3) -> (Vec3, Vec3,
Vec3) {
    // Forward points from eye to target
    let forward = (target - eye).normalize();

    // Right is perpendicular to forward and world up
    let right = forward.cross(world_up).normalize();

    // True up is perpendicular to right and forward
    let up = right.cross(forward);

    (right, up, forward)
}

fn main() {
    let eye = Vec3::new(0.0, 0.0, 5.0);      // Camera position
    let target = Vec3::new(0.0, 0.0, 0.0);   // Looking at origin
    let world_up = Vec3::unit_y();           // Y is up

    let (right, up, forward) = look_at_basis(eye, target, world_up);

    println!("Camera basis:");
    println!("  Right:   {}", right);
    println!("  Up:      {}", up);
    println!("  Forward: {}", forward);

    // Verify orthogonality
    println!("\nOrthogonality check:");
    println!("  right · up = {}", right.dot(up));
    println!("  right · forward = {}", right.dot(forward));
    println!("  up · forward = {}", up.dot(forward));
}
```

Output:

```
Camera basis:
  Right:   (1, 0, 0)
  Up:      (0, 1, 0)
  Forward: (0, 0, -1)

Orthogonality check:
  right · up = 0
  right · forward = 0
  up · forward = 0
```

Practical Example: Torque in Physics

In physics, torque (rotational force) is the cross product of the position vector and force:

$$\vec{\tau} = \vec{r} \times \vec{F}$$

```
fn main() {
    // Wrench 0.5m from the bolt
    let position = Vec3::new(0.5, 0.0, 0.0);  // 0.5m along x

    // Force of 100N straight down
    let force = Vec3::new(0.0, -100.0, 0.0);

    let torque = position.cross(force);
    println!("Torque: {}", torque);  // (0, 0, -50)
    println!("Torque magnitude: {} N·m", torque.length());  // 50

    // Force at an angle (less effective)
    let angled_force = Vec3::new(-50.0, -86.6, 0.0);  // 60° from horizontal
    let torque2 = position.cross(angled_force);
    println!("Angled torque: {} ({:.1} N·m)", torque2, torque2.length());
}
```

Practical Example: Determining Winding Order

In graphics, triangles have a **winding order** (clockwise or counter-clockwise). The cross product tells you which side is “front”:

```
fn is_front_facing(p0: Vec3, p1: Vec3, p2: Vec3, view_dir: Vec3) -> bool {
    let normal = (p1 - p0).cross(p2 - p0);
    normal.dot(view_dir) < 0.0  // Normal points toward viewer
}

fn main() {
    let view_dir = Vec3::new(0.0, 0.0, -1.0);  // Looking into screen

    // Counter-clockwise triangle (front-facing)
    let p0 = Vec3::new(0.0, 0.0, 0.0);
    let p1 = Vec3::new(1.0, 0.0, 0.0);
    let p2 = Vec3::new(0.0, 1.0, 0.0);

    println!("CCW triangle front-facing? {}", is_front_facing(p0, p1, p2,
view_dir));

    // Clockwise (swap p1 and p2)
    println!("CW triangle front-facing? {}", is_front_facing(p0, p2, p1,
view_dir));
}
```

Try It

1. **Scalar triple product:** The volume of a parallelepiped is:

$$V = |\vec{a} \cdot (\vec{b} \times \vec{c})|$$

```rust
pub fn triple_product(a: Vec3, b: Vec3, c: Vec3) -> f64 {
    a.dot(b.cross(c))
}
```

2. **Check if three points are collinear:**

```rust
pub fn are_collinear(p0: Vec3, p1: Vec3, p2: Vec3, epsilon: f64) -> bool
{
    let v1 = p1 - p0;
    let v2 = p2 - p0;
    v1.cross(v2).length_squared() < epsilon * epsilon
}
```

3. **Reflect a vector across a plane** with normal $\hat{n}$:

$$\vec{r} = \vec{v} - 2(\vec{v} \cdot \hat{n})\hat{n}$$

   (This uses dot product, not cross product, but is related to normal vectors.)

4. **Tests:**

```rust
#[test]
fn test_cross_anticommutative() {
    let a = Vec3::new(1.0, 2.0, 3.0);
    let b = Vec3::new(4.0, 5.0, 6.0);
    assert_eq!(a.cross(b), -(b.cross(a)));
}

#[test]
fn test_cross_perpendicular() {
    let a = Vec3::new(1.0, 2.0, 3.0);
    let b = Vec3::new(4.0, 5.0, 6.0);
    let c = a.cross(b);
    assert!(a.dot(c).abs() < 1e-10);
    assert!(b.dot(c).abs() < 1e-10);
}

#[test]
fn test_cross_self_is_zero() {
    let a = Vec3::new(1.0, 2.0, 3.0);
    assert_eq!(a.cross(a), Vec3::zero());
}

#[test]
fn test_unit_vector_cross_products() {
    let i = Vec3::unit_x();
    let j = Vec3::unit_y();
    let k = Vec3::unit_z();
    assert_eq!(i.cross(j), k);
    assert_eq!(j.cross(k), i);
    assert_eq!(k.cross(i), j);
}
```

Chapter Summary

You've built a complete 3D vector type with all the fundamental operations:

Operation	Math	Code
Create	$\vec{v} = (x, y, z)$	`Vec3::new(x, y, z)`
Add	$\vec{a} + \vec{b}$	`a + b`
Subtract	$\vec{a} - \vec{b}$	`a - b`

Operation	Math	Code
Scale	$c\vec{v}$	`c * v` or `v * c`
Dot product	$\vec{a} \cdot \vec{b}$	`a.dot(b)`
Length	$\|\vec{v}\|$	`v.length()`
Normalize	$\hat{v}$	`v.normalize()`
Cross product	$\vec{a} \times \vec{b}$	`a.cross(b)`

These operations are the foundation of:

- **Computer graphics**: Positions, directions, normals, lighting
- **Physics simulations**: Velocity, acceleration, forces, torque
- **Machine learning**: Feature vectors, embeddings
- **Geometry**: Distances, angles, areas, volumes

Complete `vec3.rs`

Here's the complete module with everything from Chapter 1:

```rust
// src/vec3.rs

use std::fmt;
use std::ops::{Add, Sub, Mul, Div, Neg, AddAssign, SubAssign, MulAssign,
DivAssign};

/// A 3-dimensional vector with f64 components.
#[derive(Debug, Clone, Copy, PartialEq)]
pub struct Vec3 {
    pub x: f64,
    pub y: f64,
    pub z: f64,
}

impl Vec3 {
    pub fn new(x: f64, y: f64, z: f64) -> Self {
        Vec3 { x, y, z }
    }

    pub fn zero() -> Self {
        Vec3 { x: 0.0, y: 0.0, z: 0.0 }
    }

    pub fn unit_x() -> Self {
        Vec3 { x: 1.0, y: 0.0, z: 0.0 }
    }

    pub fn unit_y() -> Self {
        Vec3 { x: 0.0, y: 1.0, z: 0.0 }
    }

    pub fn unit_z() -> Self {
        Vec3 { x: 0.0, y: 0.0, z: 1.0 }
    }

    pub fn dot(self, other: Vec3) -> f64 {
        self.x * other.x + self.y * other.y + self.z * other.z
    }

    pub fn cross(self, other: Vec3) -> Vec3 {
        Vec3 {
            x: self.y * other.z - self.z * other.y,
            y: self.z * other.x - self.x * other.z,
            z: self.x * other.y - self.y * other.x,
        }
    }

    pub fn length_squared(self) -> f64 {
        self.dot(self)
    }

    pub fn length(self) -> f64 {
        self.length_squared().sqrt()
    }

    pub fn normalize(self) -> Vec3 {
```

```
        let len = self.length();
        if len == 0.0 { Vec3::zero() } else { self / len }
    }

    pub fn try_normalize(self) -> Option<Vec3> {
        let len = self.length();
        if len == 0.0 { None } else { Some(self / len) }
    }

    pub fn is_normalized(self, epsilon: f64) -> bool {
        (self.length_squared() - 1.0).abs() < epsilon
    }

    pub fn distance(self, other: Vec3) -> f64 {
        (other - self).length()
    }

    pub fn distance_squared(self, other: Vec3) -> f64 {
        (other - self).length_squared()
    }

    pub fn angle_between(self, other: Vec3) -> f64 {
        let dot = self.dot(other);
        let lengths = self.length() * other.length();
        if lengths == 0.0 { return 0.0; }
        (dot / lengths).clamp(-1.0, 1.0).acos()
    }

    pub fn lerp(self, other: Vec3, t: f64) -> Vec3 {
        self * (1.0 - t) + other * t
    }

    pub fn is_perpendicular(self, other: Vec3, epsilon: f64) -> bool {
        self.dot(other).abs() < epsilon
    }

    pub fn is_parallel(self, other: Vec3, epsilon: f64) -> bool {
        self.cross(other).length_squared() < epsilon * epsilon
    }
}

impl fmt::Display for Vec3 {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "({}, {}, {})", self.x, self.y, self.z)
    }
}

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

impl Sub for Vec3 {
    type Output = Vec3;
```

```
    fn sub(self, other: Vec3) -> Vec3 {
        Vec3 { x: self.x - other.x, y: self.y - other.y, z: self.z - other.z
}
    }
}

impl Neg for Vec3 {
    type Output = Vec3;
    fn neg(self) -> Vec3 {
        Vec3 { x: -self.x, y: -self.y, z: -self.z }
    }
}

impl Mul<f64> for Vec3 {
    type Output = Vec3;
    fn mul(self, scalar: f64) -> Vec3 {
        Vec3 { x: self.x * scalar, y: self.y * scalar, z: self.z * scalar }
    }
}

impl Mul<Vec3> for f64 {
    type Output = Vec3;
    fn mul(self, vec: Vec3) -> Vec3 {
        Vec3 { x: self * vec.x, y: self * vec.y, z: self * vec.z }
    }
}

impl Div<f64> for Vec3 {
    type Output = Vec3;
    fn div(self, scalar: f64) -> Vec3 {
        Vec3 { x: self.x / scalar, y: self.y / scalar, z: self.z / scalar }
    }
}

impl AddAssign for Vec3 {
    fn add_assign(&mut self, other: Vec3) {
        self.x += other.x; self.y += other.y; self.z += other.z;
    }
}

impl SubAssign for Vec3 {
    fn sub_assign(&mut self, other: Vec3) {
        self.x -= other.x; self.y -= other.y; self.z -= other.z;
    }
}

impl MulAssign<f64> for Vec3 {
    fn mul_assign(&mut self, scalar: f64) {
        self.x *= scalar; self.y *= scalar; self.z *= scalar;
    }
}

impl DivAssign<f64> for Vec3 {
    fn div_assign(&mut self, scalar: f64) {
        self.x /= scalar; self.y /= scalar; self.z /= scalar;
    }
}
```

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_new() {
        let v = Vec3::new(1.0, 2.0, 3.0);
        assert_eq!(v.x, 1.0);
        assert_eq!(v.y, 2.0);
        assert_eq!(v.z, 3.0);
    }

    #[test]
    fn test_add() {
        let a = Vec3::new(1.0, 2.0, 3.0);
        let b = Vec3::new(4.0, 5.0, 6.0);
        assert_eq!(a + b, Vec3::new(5.0, 7.0, 9.0));
    }

    #[test]
    fn test_dot() {
        let a = Vec3::new(1.0, 2.0, 3.0);
        let b = Vec3::new(4.0, 5.0, 6.0);
        assert_eq!(a.dot(b), 32.0);
    }

    #[test]
    fn test_cross() {
        let i = Vec3::unit_x();
        let j = Vec3::unit_y();
        assert_eq!(i.cross(j), Vec3::unit_z());
    }

    #[test]
    fn test_length() {
        let v = Vec3::new(3.0, 4.0, 0.0);
        assert_eq!(v.length(), 5.0);
    }

    #[test]
    fn test_normalize() {
        let v = Vec3::new(3.0, 4.0, 0.0);
        let n = v.normalize();
        assert!((n.length() - 1.0).abs() < 1e-10);
    }
}
```

What's Next?

In Chapter 2, we'll build a `Matrix` type to represent linear transformations—rotations, scaling, shearing, and more. Matrices take everything we've learned about vectors and add the power to transform entire coordinate systems.