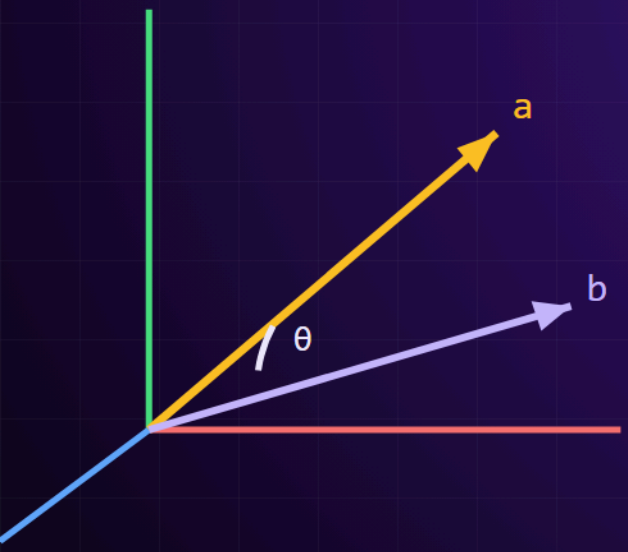


$$a \cdot b = |a|\,|b| \cos\theta$$

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$q = w + xi + yj + zk$$

$$v' = M \cdot v$$

VECTORS · MATRICES · QUATERNIONS

3D Math for JavaScript Developers

The math behind positions, rotations, cameras, and motion — explained with code

Vectors, matrices, quaternions, curves, and physics — with exercises and solutions in every chapter

VECTORS · MATRICES · QUATERNIONS · CAMERAS

THENEXBOOKS

3D Math for JavaScript Developers

Vectors, Matrices, Quaternions, and Transforms with Three.js
The Math Behind Positions, Rotations, Cameras, Curves, and Motion — Explained with Code

First Edition

THENEXBOOKS

3D Math for JavaScript Developers
Vectors, Matrices, Quaternions, and Transforms with Three.js

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples and exercise solutions may be used freely for learning and in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Libraries such as Three.js release new versions frequently; always consult the official documentation for the versions you use.

Three.js, React, and other product names are trademarks of their respective owners. Their use in this book is for identification only; this book is independent and is not affiliated with or endorsed by the maintainers of any of these projects.

All organizations, people, places, and products in the examples are fictional. Any resemblance to real ones is coincidental.

Preface

Sooner or later every 3D developer hits a wall made of math. An object rotates the wrong way, a camera points somewhere unexpected, a model appears in the wrong place, or motion jitters as the frame rate changes. Libraries such as Three.js hide much of the math, but they cannot hide it completely — and when something goes wrong, understanding is the only reliable fix.

This book teaches the math you actually need for 3D on the web, with code instead of proofs. Every idea is introduced with a picture, written as a small JavaScript function so you can see exactly what it does, checked with real numbers, and then connected to the Three.js classes you will use every day: Vector3, Matrix4, Quaternion, Euler, Plane, Ray, Box3, and the curve classes. High-school algebra is enough to start.

Every chapter ends with graded exercises and worked solutions. By the end you will be able to reason about positions, directions, rotations, cameras, intersections, curves, and motion with confidence — and you will have built a camera rig, a solar system, and a drone flight simulator to prove it.

Who This Book Is For

- **JavaScript developers using Three.js** who want to understand what their code is doing.
- **Front-end and creative developers** moving from 2D to 3D.
- **Game and simulation hobbyists** who need reliable movement, cameras, and collisions.
- **Students** who want an applied, code-first companion to linear algebra.

What You Will Learn

- Coordinate systems, vectors, and the dot and cross products.
- Matrices, transforms, the scene graph, and local versus world space.
- Euler angles, gimbal lock, quaternions, and smooth interpolation.
- Cameras, projection, rays, and intersections with lines, planes, and volumes.
- Normals, curves, procedural geometry, motion, collisions, springs, and noise.
- Floating-point pitfalls and three complete projects.

How This Book Is Organized

Part I — Foundations (Chapters 1–5): Why 3D Math Matters for Web Developers; Coordinate Systems, Axes, and Units; Vectors I; Vectors II; Angles, Trigonometry, and Radians. **Part II — Transforms** (Chapters 6–10): Matrices and Linear Transformations; Translation, Rotation, and Scale with 4×4 Matrices; Composing Transforms and the Scene Graph; Local Space, World Space, and Converting Between Them; Euler Angles, Rotation Order, and Gimbal Lock. **Part III — Rotations and Cameras** (Chapters 11–15): Quaternions Made Practical; Interpolation; Cameras; Projection;

From 3D to Screen and Back. **Part IV — Geometry** (Chapters 16–20): Lines, Planes, and Intersections; Bounding Volumes; Normals, Tangents, and the Math of Lighting; Curves; Procedural Geometry with Math. **Part V — Motion and Physics** (Chapters 21–25): Velocity, Acceleration, and Numerical Integration; Collision Detection Basics; Springs, Damping, and Smooth Following; Noise and Randomness for 3D; Floating-Point Precision and Numerical Pitfalls. **Part VI — Projects and Case Study** (Chapters 26–28): project; project; case study. The appendices provide a cheat sheet, templates and references, a checklist, and a glossary.

The Running Example

Throughout the book we help a small studio build a **flight simulator for Kestrel**, a fictional delivery drone flying over a small valley with a landing pad and three delivery points. Each chapter solves one of its math problems, and the final chapter assembles the complete simulator.

Conventions

Code appears in `monospace`. Examples use modern JavaScript modules and current library releases; file captions such as `src/main.js` show where code belongs. Exercises are labeled **[Easy]**, **[Medium]**, or **[Challenge]**.

> **TIP**
> A recommended practice, or a "Try It" prompt to pause and experiment.

> **NOTE**
> Background information or a useful aside.

> **WARNING**
> A common mistake that breaks rendering, wastes performance, or confuses users.

Contents

Preface

PART I — FOUNDATIONS

1 Why 3D Math Matters for Web Developers

2 Coordinate Systems, Axes, and Units

3 Vectors I: Points, Directions, Addition, and Length

4 Vectors II: Dot and Cross Products

5 Angles, Trigonometry, and Radians

PART II — TRANSFORMS

6 Matrices and Linear Transformations

7 Translation, Rotation, and Scale with 4×4 Matrices

8 Composing Transforms and the Scene Graph

9 Local Space, World Space, and Converting Between Them

10 Euler Angles, Rotation Order, and Gimbal Lock

PART III — ROTATIONS AND CAMERAS

11 Quaternions Made Practical

12 Interpolation: Lerp, Slerp, and Easing

13 Cameras: View Matrices and lookAt

14 Projection: Perspective and Orthographic

15 From 3D to Screen and Back: Projecting, Unprojecting, and Rays

PART IV — GEOMETRY

16 Lines, Planes, and Intersections

17 Bounding Volumes: Boxes and Spheres

18 Normals, Tangents, and the Math of Lighting

19 Curves: Bézier, Catmull-Rom, and Splines

20 Procedural Geometry with Math

PART V — MOTION AND PHYSICS

21 Velocity, Acceleration, and Numerical Integration

22 Collision Detection Basics

23 Springs, Damping, and Smooth Following

24 Noise and Randomness for 3D

25 Floating-Point Precision and Numerical Pitfalls

PART VI — PROJECTS AND CASE STUDY

26 Project: A Third-Person Camera Rig

27 Project: A Solar System with Orbits

28 Case Study: The Kestrel Drone Flight Simulator

APPENDICES

A 3D Math Cheat Sheet

B Formulas and Three.js Math API Reference

C Debugging Transforms Checklist

D Glossary

PART I

Foundations

Why 3D Math Matters for Web Developers, Coordinate Systems, Axes, and Units, Vectors I, Vectors II, Angles, Trigonometry, and Radians.

1 · Why 3D Math Matters for Web Developers
2 · Coordinate Systems, Axes, and Units
3 · Vectors I
4 · Vectors II
5 · Angles, Trigonometry, and Radians

CHAPTER 1 · PART I — FOUNDATIONS

Why 3D Math Matters for Web Developers

Every Three.js bug that makes you say "why is it facing that way?" is a math question in disguise. In this chapter we name those questions, meet the drone simulator we will build math for throughout the book, and run our first real calculations in Node and in the browser.

The Bugs That Send You to the Math

Most JavaScript developers arrive at 3D the same way. You follow a getting-started tutorial, a green cube spins on screen, and everything feels easy. Then you try to build something of your own and the cube starts misbehaving in ways that no amount of trial and error seems to fix. You change a sign, the problem moves somewhere else. You swap two lines, and the object now spins around the wrong point. You copy a snippet from a forum answer, and it works until the camera moves.

Here are the symptoms we hear about most often, and the idea that actually explains each one:

What you see	The math underneath	Where we fix it
A character moves faster when you hold two arrow keys	Vector length and normalization	Chapter 3 (and this chapter)
An enemy cannot tell if the player is in front of it or behind	The dot product	Chapter 4
A rotation spins around the wrong axis, or flips at 90 degrees	Rotation order and gimbal lock	Chapters 10 and 11
A child object ends up in the wrong place when its parent moves	Local space versus world space	Chapters 8 and 9
The camera looks the wrong way or rolls sideways	View matrices and `lookAt`	Chapter 13
Clicking the screen selects the wrong object	Projection, unprojection, and rays	Chapters 14 and 15
Motion speeds up on a fast monitor	Time steps and integration	Chapter 21 (and this chapter)
Far-away objects jitter or flicker	Floating-point precision	Chapter 25

None of these problems require advanced mathematics. They require a handful of ideas — vectors, matrices, quaternions, and the notion of a coordinate space — understood well enough that you can predict what a line of code will do *before* you run it. That is the goal of this book: not to turn you into a

mathematician, but to make the numbers in your scene stop being mysterious.

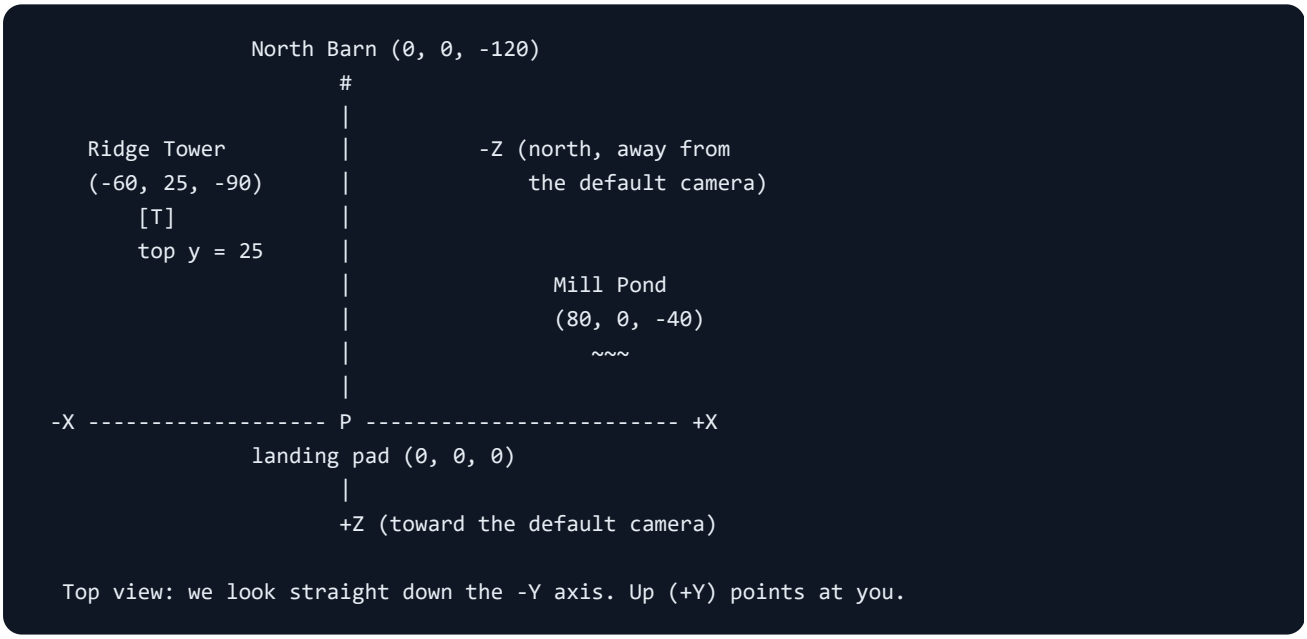
High-school algebra is enough to start. If you can read `x = 3 + 4` and remember that the square root of 25 is 5, you have all the background you need. Everything else we build step by step, always in the same order: first a tiny plain-JavaScript function where you can see every multiplication, then the equivalent Three.js call that you will use in real projects.

Meet Kestrel and Alder Valley

A book about math needs concrete numbers, so we will spend the whole book working on one project. **Fieldline Labs**, a small fictional studio, is building a browser-based flight simulator for its delivery drone, **Kestrel**. The simulator lets pilots practice flying over **Alder Valley**, a small fictional valley with a landing pad, three delivery points, and a wind sock.

You are the new developer on the team. **Nadia Brooks**, the lead engineer, owns the flight model and the camera system. **Leo Park**, the gameplay programmer, owns delivery missions, scoring, and the heads-up display. Throughout the book they will hand you small tasks — "make the drone face the next delivery point", "tell me if the barn is visible from the camera", "smooth out this landing" — and each task will need one new piece of math.

Here is the valley seen from directly above. The landing pad sits at the origin, and the three delivery points are scattered around it:



Ridge Tower is the only delivery point above the ground: its drop-off platform sits at the top of the tower, 25 meters up. That single detail already makes some of our calculations three-dimensional rather than flat.

The conventions we use everywhere

3D math is full of choices that are arbitrary but must be made consistently: which way is up, what one unit means, whether angles are in degrees or radians. Mixing conventions is the single most common source of "almost right" bugs. We fix them once, here, and keep them for the entire book:

Topic	Convention in this book
Coordinate system	Three.js right-handed coordinates
Up direction	+Y is up
Default camera view	The camera looks down the –Z axis
Distance	1 unit = 1 meter
Angles in code	Radians (degrees only in prose)
Time	Seconds
Kestrel limits	Max speed 15 m/s, max climb rate 4 m/s

Chapter 2 explains what "right-handed" means and why +Y is up; Chapter 5 covers radians. For now, just remember the table: when you see the number 15 in a Kestrel example, it means fifteen meters per second, and when you see –120 on the z axis, it means 120 meters to the north of the pad.

What "3D Math" Really Means for a Web Developer

The phrase "3D math" sounds like one enormous subject. In practice, the math you need for interactive 3D on the web is a fairly small toolbox, and every tool in it answers a specific kind of question:

- **Vectors** answer "where?" and "which way?" — positions, directions, velocities, and the distance between things (Chapters 3 and 4).
- **Angles and trigonometry** answer "how much turn?" — headings, orbits, field of view (Chapter 5).
- **Matrices** answer "how do I move, turn, and resize many points at once?" — and how to chain those operations (Chapters 6 to 9).
- **Euler angles and quaternions** answer "how is this object oriented, and how do I turn it smoothly?" (Chapters 10 to 12).
- **Cameras and projection** answer "how does a 3D point become a pixel, and how does a mouse click become a 3D ray?" (Chapters 13 to 15).
- **Geometry** answers "does this hit that?", "which way does this surface face?", and "how do I build shapes from formulas?" (Chapters 16 to 20).
- **Motion** answers "where will it be next frame?" — velocity, springs, collisions, noise, and the limits of floating-point numbers (Chapters 21 to 25).

All of these tools serve one pipeline. Every frame, Three.js takes the vertices of every mesh, moves them through a chain of coordinate spaces, and finally turns them into pixels. You do not have to implement this pipeline — the library and the GPU do it — but you need to know it exists, because every bug in the table above is a value going wrong at one of its stages:

```
  model space          world space          camera space        screen
 (vertices as     --> (where things   --> (relative to   --> (pixels on
  the artist made      are in the          the camera's        the canvas)
  them)                valley)             eye)
        |                    |                   |                  |
   object's matrix      camera's view       projection         viewport
   (Chapters 7-9)       matrix (Ch. 13)     matrix (Ch. 14)    (Ch. 15)
```

By the end of the book you will be able to point at any stage of this diagram and say what numbers live there and how they got there. Chapter 28 ties the whole pipeline together in a case study of the finished Kestrel simulator.

Your First Calculation: How Far Is Each Delivery Point?

Leo's first request is simple: the mission screen should show how far each delivery point is from the landing pad, and the minimum time Kestrel needs to get there at full speed. This is the distance formula you may remember from school, extended to three dimensions. For two points a and b:

`distance = sqrt((bx − ax)² + (by − ay)² + (bz − az)²)`

In words: find how far apart the points are along each axis, square those three differences, add them, and take the square root. Chapter 3 shows exactly why this works; for now let us use it. We store points as plain arrays `[x, y, z]` so that nothing is hidden:

distances.mjs

```
// Plain JavaScript: points are arrays [x, y, z], distances in meters.
const places = {
  northBarn: [0, 0, -120],
  millPond: [80, 0, -40],
  ridgeTop: [-60, 25, -90],
};
const pad = [0, 0, 0];
const MAX_SPEED = 15; // m/s

function distance(a, b) {
  const dx = b[0] - a[0];
  const dy = b[1] - a[1];
  const dz = b[2] - a[2];
  return Math.sqrt(dx * dx + dy * dy + dz * dz);
}

const round3 = (n) => Math.round(n * 1000) / 1000;

for (const [name, point] of Object.entries(places)) {
```

```
  const d = distance(pad, point);
  console.log(name, round3(d), 'm', round3(d / MAX_SPEED), 's');
}
```

Run it with `node distances.mjs`. We round to three decimals for display throughout the book, and every number printed in these chapters was produced by running the code shown:

```
northBarn 120 m 8 s
millPond 89.443 m 5.963 s
ridgeTop 111.018 m 7.401 s
```

North Barn is the easy case: it lies straight along the z axis, so the distance is just 120. Mill Pond needs the formula: `sqrt(80² + 0² + (−40)²) = sqrt(8000) ≈ 89.443`. Ridge Tower uses all three axes: `sqrt((−60)² + 25² + (−90)²) = sqrt(12325) ≈ 111.018`. Notice that the tower's height adds to the distance — a flat map would underestimate it.

Now the same thing with Three.js. The `Vector3` class stores `x`, `y`, and `z`, and has a `distanceTo()` method that does the arithmetic for us. Three.js math classes work fine in Node, so you can test them without a browser:

distances-three.mjs

```
import * as THREE from 'three';

const pad = new THREE.Vector3(0, 0, 0);
const ridgeTop = new THREE.Vector3(-60, 25, -90);

console.log(pad.distanceTo(ridgeTop).toFixed(3)); // 111.018
```

Same answer, less typing. This is the pattern for the whole book: the plain function shows you the math, and the Three.js method is what you write once you trust it.

> **TRY IT**
>
> Add a fourth entry to `places` for a point of your own, such as a hilltop at `[40, 30, -100]`. Before running the script, estimate the distance in your head (it must be more than 100 m because z alone is −100). Then run it and compare.

Bug Number One: The Diagonal Speed Boost

Here is a bug almost every game developer ships at least once. Leo wires the keyboard to Kestrel: the right arrow adds 1 to the x direction, the up arrow adds −1 to z (north). Then he multiplies by the max speed:

```
// Leo's first attempt
const input = [0, 0, 0];
if (keys.ArrowRight) input[0] += 1;
```

```
if (keys.ArrowUp) input[2] -= 1;

const velocity = input.map((c) => c * 15); // m/s
```

Pressing only the up arrow gives a velocity of `(0, 0, -15)` — exactly 15 m/s. Pressing both gives `(15, 0, -15)`. How fast is that? Speed is the length of the velocity, so we use the same square-root formula: `sqrt(15² + 15²) = sqrt(450) ≈ 21.213` m/s. Pilots discover within minutes that flying diagonally is about 41 percent faster than the drone's real maximum. In a flight simulator that is not a quirk; it is wrong physics.

The input vector `(1, 0, -1)` has a length of about 1.414, not 1. The fix is to **normalize** it — divide each component by the length so the result has length exactly 1 — and only then multiply by the speed:

```
function normalize(v) {
  const len = Math.sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
  if (len === 0) return [0, 0, 0]; // no keys pressed: no direction
  return [v[0] / len, v[1] / len, v[2] / len];
}

const direction = normalize(input);              // [0.707, 0, -0.707]
const velocity = direction.map((c) => c * 15);   // [10.607, 0, -10.607]
```

Now the diagonal velocity is `(10.607, 0, -10.607)`, and its length is exactly 15. Each axis gets less than 15 on its own, but the combined speed is correct. In Three.js the same fix reads naturally:

```
const velocity = new THREE.Vector3(1, 0, -1).normalize().multiplyScalar(15);
console.log(velocity.length()); // 15
```

Chapter 3 covers normalization properly, including the zero-length case guarded above. What matters right now is the lesson: a *direction* and a *speed* are different things, and mixing them silently changes your numbers.

Bug number two: speed that depends on the monitor

A related bug hides in the animation loop. If you move Kestrel a fixed amount every frame — say 0.25 m — then its speed depends on how many frames per second the browser draws. At 60 frames per second, 0.25 m per frame is 15 m/s. On a 144 Hz monitor the same code gives 0.25 × 144 = 36 m/s; on a struggling laptop drawing 30 frames per second it gives 7.5 m/s. The fix is to scale by elapsed time: `position += velocity × dt`, where `dt` is the seconds since the last frame. We will use that pattern from Chapter 3 onward and study it properly in Chapter 21.

How Three.js Math Classes Behave

Three.js ships its own small math library: `Vector2`, `Vector3`, `Vector4`, `Matrix3`, `Matrix4`, `Quaternion`, `Euler`, `Box3`, `Sphere`, `Plane`, `Ray`, and a collection of helpers in `THREE.MathUtils`. Every `Object3D` — meshes, cameras, lights, groups — carries a `position` (a `Vector3`), a `rotation` (an `Euler`), a `quaternion`, a `scale`, and several `Matrix4` objects. When you write `mesh.position.x = 5` you are already doing vector math.

These classes have one design decision that surprises nearly every JavaScript developer: **most methods change the object they are called on**, and then return that same object so you can chain calls. Coming from array methods like `map()` and `concat()`, which return new values, you might expect `a.sub(b)` to return a new vector. It does not. Watch:

```
const northBarn = new THREE.Vector3(0, 0, -120);
const millPond = new THREE.Vector3(80, 0, -40);

// "The vector from Mill Pond to North Barn"... or so we hope.
const toBarn = northBarn.sub(millPond);

console.log(toBarn);              // Vector3 { x: -80, y: 0, z: -80 }
console.log(northBarn);           // Vector3 { x: -80, y: 0, z: -80 }  (!)
console.log(toBarn === northBarn); // true
```

The answer in `toBarn` is correct, but North Barn has just moved to `(-80, 0, -80)`. If `northBarn` were the position of the barn mesh, the barn would visibly jump across the valley on the next frame. There are three standard ways to avoid this:

```
// 1. Clone first: make a copy, then change the copy.
const toBarn1 = northBarn.clone().sub(millPond);

// 2. Use the two-argument form: write a - b into a target vector.
const toBarn2 = new THREE.Vector3().subVectors(northBarn, millPond);

// 3. Reuse a scratch vector you created once (best inside animation loops).
const _tmp = new THREE.Vector3();
function vectorFromTo(a, b) {
  return _tmp.subVectors(b, a); // caller must use or copy it right away
}
```

All three give `(-80, 0, -80)`, a vector about 113.137 m long, without touching the originals. The first is easiest to read. The third avoids creating a new object every frame, which reduces garbage-collection work in a busy animation loop; it is the style you will find inside Three.js itself. Use it carefully: whatever the function returns will be overwritten by the next call.

> **WARNING**
>
> Never call a mutating method directly on `mesh.position`, `camera.position`, or any vector you did not create for scratch work, unless you really mean to move that object. `camera.position.sub(target)` does not "compute the offset to the target" — it teleports the camera.

Our plain-JavaScript functions take the opposite approach: they never change their inputs and always return new arrays. That makes them easy to reason about and easy to test, which is exactly what we want while learning. Real Three.js code trades a little of that clarity for speed.

Setting Up Your Workbench

You need two environments for this book: Node for checking numbers and a Vite project for seeing them on screen.

Node for math experiments

Install a current Node LTS release. Make a folder, initialize it, and install Three.js:

```
mkdir kestrel-math && cd kestrel-math
npm init -y
npm install three
```

Save files with the `.mjs` extension (or add `"type": "module"` to `package.json`) so that `import * as THREE from 'three'` works. Three.js's math classes do not need a browser or a GPU, so a Node script is the fastest way to check any calculation in this book.

Vite for rendering

For anything you want to look at, create a Vite project with the vanilla JavaScript template and add Three.js:

```
npm create vite@latest alder-valley -- --template vanilla
cd alder-valley
npm install three
npm run dev
```

Replace the contents of `src/main.js` with the scene below, and make sure the page's CSS removes the default body margin so the canvas fills the window. This is our "map view" of Alder Valley: a ground grid, colored axes at the origin, and a simple box marker at each place we know.

src/main.js

```
import * as THREE from 'three';
import { OrbitControls } from 'three/addons/controls/OrbitControls.js';

const renderer = new THREE.WebGLRenderer({ antialias: true });
renderer.setSize(innerWidth, innerHeight);
renderer.setPixelRatio(Math.min(devicePixelRatio, 2));
document.body.appendChild(renderer.domElement);

const scene = new THREE.Scene();
scene.background = new THREE.Color(0xbfd8ea);

const camera = new THREE.PerspectiveCamera(50, innerWidth / innerHeight, 0.1, 1000);
camera.position.set(150, 120, 110);
const controls = new OrbitControls(camera, renderer.domElement);
controls.target.set(0, 0, -60);
controls.update();

scene.add(new THREE.HemisphereLight(0xffffff, 0x556644, 2));
scene.add(new THREE.GridHelper(400, 40, 0x335533, 0x88aa88)); // 10 m squares
scene.add(new THREE.AxesHelper(30));                          // 30 m axes

function marker(color, x, y, z, height = 4) {
  const mesh = new THREE.Mesh(
    new THREE.BoxGeometry(6, height, 6),
    new THREE.MeshStandardMaterial({ color })
  );
  mesh.position.set(x, y + height / 2, z); // box sits ON the ground
  scene.add(mesh);
  return mesh;
}

marker(0xffffff, 0, 0, 0, 1);       // landing pad
marker(0xc0392b, 0, 0, -120);       // North Barn
marker(0x2e86c1, 80, 0, -40);       // Mill Pond
marker(0x7d3c98, -60, 0, -90, 25);  // Ridge Tower, top at y = 25

renderer.setAnimationLoop(() => {
  controls.update();
  renderer.render(scene, camera);
});
```

When the page loads you should see a pale blue sky over a large green-lined grid seen from above and to the side. Near the middle of the view, a thin white slab marks the landing pad, with three short colored lines growing out of it: red along +X, green straight up along +Y, and blue along +Z toward you. A small red box sits far away in the north (North Barn), a blue box sits off to the right (Mill Pond), and a tall purple column rises on the left (Ridge Tower). Drag with the mouse to orbit around the valley.

One line deserves attention already: `mesh.position.set(x, y + height / 2, z)`. A `BoxGeometry` is centered on its own origin, so placing a 25 m tall box at `y = 0` would bury half of it underground. Lifting it by half its height puts its base on the ground and its top at exactly 25 m. This is our first small

encounter with the difference between an object's own coordinates and the world's — the subject of Chapter 9.

> **NOTE**
>
> The `AxesHelper` and `GridHelper` are the two most useful debugging tools in Three.js. Keep them in every scene while you are learning. A sister book in this series, *Three.js from Zero*, covers scene setup in depth; here we only need enough scene to see our math.

> **TRY IT**
>
> Change the camera position to `(0, 300, 0.01)` and the target to `(0, 0, 0)`. You now look almost straight down, and the view should match the top-view ASCII map earlier in this chapter: North Barn toward the top of the screen, Mill Pond to the right, Ridge Tower upper left.

Math Hiding in Code You Already Write

Even the simplest Three.js scene is full of math, most of it quietly done for you. Look again at a few familiar lines and notice what each one really says:

- `mesh.position.set(80, 0, -40)` stores a **vector**: "this object's origin is 80 m east and 40 m north of the world origin". Chapter 3 is about exactly these three numbers.
- `mesh.rotation.y = Math.PI / 2` stores an **angle in radians** around the y axis — a quarter turn. Why a positive value turns the object counterclockwise when seen from above is a question of handedness (Chapter 2) and trigonometry (Chapter 5).
- `mesh.scale.set(2, 2, 2)` multiplies every vertex coordinate by 2. Combined with position and rotation, it becomes a **4×4 matrix** (Chapter 7).
- `group.add(mesh)` makes the mesh's position relative to the group. That one call creates a **parent-child transform chain** (Chapter 8).
- `camera.lookAt(0, 0, 0)` builds a **rotation** from a direction and an up vector, using the cross product you will meet in Chapter 4.

Three.js turns position, rotation, and scale into a matrix for every object, multiplies the matrices of parents and children together, and hands the results to the GPU. You rarely need to do that by hand, but you will constantly need to *reason* about it: to ask "where is this point in the world?" or "which way is the drone's nose pointing right now?". That reasoning is what this book trains.

A small debugging habit

Printing a `Vector3` with `console.log` shows long floating-point tails such as `-10.606601717798211`, which are hard to compare at a glance. A tiny helper makes logs readable and is worth keeping in every project:

```
// Format any {x, y, z} object as "(x, y, z)" with 3 decimals.
function fmt(v, digits = 3) {
  return `(${v.x.toFixed(digits)}, ${v.y.toFixed(digits)}, ${v.z.toFixed(digits)})`;
}

console.log(fmt(new THREE.Vector3(1, 0, -1).normalize().multiplyScalar(15)));
// (10.607, 0.000, -10.607)
```

Because the function only reads `x`, `y`, and `z`, it works for plain objects too. When a value looks wrong on screen, log it with `fmt()` once per second rather than every frame, compare it with what you expected, and the bug usually becomes obvious.

How to Read This Book

Each chapter follows the same rhythm, and knowing it will help you move quickly:

1. **An idea in words and pictures.** We describe what a concept means geometrically, usually with an ASCII diagram you can sketch on paper.
2. **A worked numeric example.** We pick real Alder Valley numbers and compute the answer step by step. Every number has been checked by running code.
3. **A plain-JavaScript function.** Arrays or simple objects, no library, every operation visible.
4. **The Three.js equivalent.** The class and method you will use in production.
5. **What you should see.** When an example renders, we tell you exactly what appears on screen, so you know immediately whether your version works.
6. **Common mistakes, exercises, and solutions.** The exercises are where the learning really happens; do at least the Easy and Medium ones.

We write math in plain text: `|v|` means the length of vector v, `a · b` is the dot product, `a × b` the cross product, and `sqrt()` a square root. Vectors are written as `(x, y, z)`. When we say "round to three decimals" we mean for display only; the code keeps full precision.

Finally, keep a notebook or a scratch file open. Drawing a quick sketch of the axes before you write a rotation or a cross product takes ten seconds and prevents an hour of sign errors. Nadia keeps a sticky note on her monitor that says simply "Which space is this in?" — by Chapter 9 you will understand why.

Common Mistakes

- **Treating mutating methods as pure.** `a.add(b)`, `a.sub(b)`, `a.normalize()`, and `a.multiplyScalar(s)` all change `a`. Use `clone()`, the two-argument forms like `addVectors()` and `subVectors()`, or a dedicated scratch vector.
- **Using a raw input vector as a velocity.** Combined key presses produce vectors longer than 1. Normalize directions before multiplying by speed.

- **Moving a fixed amount per frame.** Frame rates vary from device to device. Multiply velocities by the elapsed time in seconds.
- **Mixing units.** If one model was authored in centimeters and another in meters, distances between them are meaningless. Chapter 2 shows how to spot and fix this.
- **Flattening the world too early.** Ignoring y gives map distances, not flight distances. Ridge Tower's platform is 25 m up, and that height matters.
- **Guessing instead of measuring.** When something looks wrong, print the vectors. `console.log(mesh.position)` is often faster than ten random sign flips.

Practice Exercises

1. **[Easy]** By hand, compute the distance from the landing pad to Mill Pond, then check it with the `distance()` function.
2. **[Easy]** Which delivery point is farthest from the pad, and what is the minimum time to reach it at Kestrel's max speed?
3. **[Easy]** Rewrite the mutation example so that `northBarn` keeps its original value, using `subVectors()`.
4. **[Easy]** In the scene from this chapter, which color is the axis line that points toward the camera's starting side of the valley (+Z)? What color points up?
5. **[Medium]** Compute the straight-line distance from Mill Pond to the top of Ridge Tower.
6. **[Medium]** With the fixed diagonal velocity `(10.607, 0, -10.607)` m/s and 60 frames per second, how far does Kestrel move along each axis in one frame?
7. **[Medium]** Leo wants a "grand tour" mission: pad → Mill Pond → North Barn → Ridge Tower top → pad. Write code that computes the total length of the route and the minimum time at max speed.
8. **[Medium]** A developer moves the drone 0.25 m per frame. What speed results on a 30 Hz device and on a 144 Hz monitor?
9. **[Challenge]** Write `nearestDeliveryPoint(position)` that returns the name and distance of the closest delivery point. Test it with Kestrel hovering at `(-20, 12, -80)`.
10. **[Challenge]** Kestrel flies in a straight line from the pad to the top of Ridge Tower at its max speed of 15 m/s. Does this break the 4 m/s climb-rate limit? Compute the vertical speed to find out.

Selected Solutions

Exercise 2

North Barn is farthest at exactly 120 m (Ridge Tower's top is 111.018 m away, Mill Pond 89.443 m). At 15 m/s the minimum flight time is `120 / 15 = 8` seconds. In practice the drone must also accelerate and brake, so real missions take longer; Chapter 21 models that.