Geometry & Trigonometry

A Complete 10-Chapter Maths for Programmers Course

Topics covered:

Angles, radians & the unit circle · triangles & practical trigonometry

Vectors, dot/cross products & rotations · 3D rotations & gimbal lock

Quaternions · coordinate transformations & intersection tests

Capstone: authoring, animating, orienting, rendering, and picking one game object

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Why Geometry & Trigonometry Matters for Programmers

Geometry & Trigonometry

Chapter 1 · Why Geometry & Trigonometry Matters for Programmers

Linear Algebra Fundamentals built vectors, matrices, and — briefly, in its own Chapter 5 — a first rotation matrix. Calculus & Optimization differentiated `sin` and `cos` without dwelling on what they actually measure. This course puts geometry itself in the foreground: the practical trigonometry and spatial reasoning behind graphics, game development, and computer vision, where a genuinely small mistake — like the one below — turns into a very visible bug.

A Real Bug: Subtracting Angles the Naive Way

A compass heading of `350°` needs to turn toward a target heading of `10°`. The obvious calculation is `target - current`:

```
current = 350   # degrees
target  = 10    # degrees
turn = target - current
print(turn)     # -340
```

VERIFIED DIRECTLY — THE NAIVE ANSWER IS WILDLY WRONG

`10 - 350 = -340` degrees. Taken literally, this tells a robot or a game character to turn **340° the wrong way around**, when the two headings are actually only `20°` apart on the compass — `350°` and `10°` are close neighbors either side of due north, not far apart. The correct, wrapped difference — computed either by the standard modulo formula `(target - current + 180) mod 360 - 180` or by the equivalent trigonometric identity `atan2(sin(Δ), cos(Δ))` — gives exactly `20°` both ways, confirmed directly.

This isn't a floating-point bug in the sense earlier Maths for Programmers courses covered — the arithmetic `10 - 350` is computed perfectly correctly. The bug is **conceptual**: angles wrap around, and ordinary subtraction doesn't know that. Recognizing where geometry has its own rules — rules that plain arithmetic doesn't automatically respect — is exactly what this course is about.

A Preview: The Dot Product Already Knows the Angle

Linear Algebra Fundamentals defined the dot product algebraically. Geometrically, it encodes an angle directly: `cos(θ) = (a·b) / (|a||b|)`.

VERIFIED DIRECTLY — RECOVERING A KNOWN ANGLE FROM TWO VECTORS

For `v1 = (1,0)` and `v2 = (1,1)`: `v1·v2 = 1`, `|v1|=1`, `|v2|=√2`, so `θ = arccos(1/√2) = 45.00000000000001°` — matching the geometrically obvious answer (a 45° diagonal) to floating-point precision. Chapter 4 builds this into a full, practical toolkit for angles-between-vectors, surface normals, and lighting calculations.

Five Concrete Connections to Real Code

GEOMETRY/TRIG TOPIC	WHERE IT ACTUALLY SHOWS UP
Angle wraparound (Ch.2)	Compass headings, joystick input, character-facing logic — exactly this chapter's own verified bug
Dot/cross products (Ch.4)	Lighting (surface normal · light direction), collision detection, determining which side of a line a point is on
Rotation matrices & quaternions (Ch.5-7)	Every camera, character, and object orientation in a 3D game engine or robotics system
Coordinate transformations (Ch.8)	The camera/projection pipeline that turns a 3D scene into 2D pixels on screen
Intersection tests (Ch.9)	Mouse-picking, ray casting, collision detection between game objects

What This Course Won't Cover

Geometry as a full mathematical field is enormous, and this course deliberately covers only what a working programmer building graphics, game, or computer-vision code actually needs:

- **Formal, proof-based Euclidean geometry** — axioms, theorems, and compass-and-straightedge constructions stay out of scope; this course treats geometry computationally and practically, not as classical proof
- **Projective and non-Euclidean geometry** — genuinely useful in specialized computer-vision and computer-graphics theory, but a deeper, more abstract topic than this course's own practical scope
- **Differential geometry and manifolds** — curvature, geodesics, and the formal machinery behind them belong to a more advanced, specialized course than this one

WHY THIS SCOPE, SPECIFICALLY

Every topic from Chapter 2 onward exists because it's a direct prerequisite for the kind of everyday spatial-reasoning code that shows up in graphics, games, and computer vision — angles, rotations, coordinate transforms, and

intersection tests. This capstone-driven course builds toward exactly that toolkit, not a general survey of geometry as a mathematical discipline.

Where This Course Is Headed

CHAPTER	TOPIC
2	Angles, Radians & the Unit Circle
3	Triangles: The Law of Sines, the Law of Cosines & Practical Trigonometry
4	Vectors & Dot/Cross Products in Geometric Context
5	2D Rotations & Rotation Matrices
6	3D Rotations: Euler Angles & Gimbal Lock
7	Quaternions
8	Coordinate Systems & Transformations
9	Geometric Primitives & Intersection Tests
10	Capstone — Building a Small 2D/3D Geometry Toolkit

THIS COURSE'S THROUGHLINE

Every chapter answers a version of the same question: given two or more objects placed somewhere in space, how do you compute the angle, distance, orientation, or intersection between them, reliably and correctly? Angles and triangles (Ch.2-3) give the basic vocabulary; vectors (Ch.4) give the tools to measure relationships between directions; rotations (Ch.5-7) give the tools to reorient objects without distorting them; coordinate transforms (Ch.8) give the tools to move between different frames of reference; and intersection tests (Ch.9) put all of it to work answering "do these two things touch?" — exactly the question a game engine or a computer-vision system asks constantly.

Hands-On Exercises

EXERCISE 1

A drone's current heading is `5°` and it needs to turn to face `340°`. Using this chapter's own naive-subtraction bug and its own wrapped-difference formula, compute both the naive result and the correct result, and explain which direction the drone should actually turn.

View solution

EXERCISE 2

Using this chapter's own dot-product angle formula, compute the angle between `v1 = (0,1)` and `v2 = (1,0)`, and explain in your own words why the dot product alone — without also considering the cross product — cannot tell you whether `v2` is rotated clockwise or counterclockwise from `v1`.

View solution

EXERCISE 3

Using this chapter's own explanation of the heading-subtraction bug, explain why it is a "conceptual" bug rather than a floating-point precision bug of the kind covered in Numerical Methods & Floating-Point Computation — what specifically would (and wouldn't) get fixed by using a more precise numeric type?

View solution

CHAPTER 1 QUICK REFERENCE

- Verified: naively subtracting two compass headings (`10° − 350°`) gives `-340°`, wildly wrong; the correct wrapped difference — via modulo or `atan2(sin Δ, cos Δ)` — is `20°`
- Angle wraparound is a **conceptual** bug, not a floating-point precision bug — ordinary arithmetic doesn't know that angles repeat every 360°
- The dot product encodes angle directly: `cos(θ)=(a·b)/(|a||b|)` — verified recovering exactly `45°` for two vectors forming a diagonal
- Five direct connections: heading/input logic, lighting/collision via dot-cross products, 3D orientation via rotations/quaternions, camera/projection pipelines, mouse-picking/collision via intersection tests
- Deliberately out of scope: formal proof-based Euclidean geometry, projective/non-Euclidean geometry, differential geometry/manifolds
- **Next chapter:** Angles, radians, and the unit circle — the vocabulary every later chapter builds on

CHAPTER 2 OF 10

Angles, Radians & the Unit Circle

Geometry & Trigonometry

Chapter 2 · Angles, Radians & the Unit Circle

Chapter 1 showed a conceptual angle-wraparound bug — ordinary subtraction not knowing that headings repeat every 360°. This chapter builds the actual vocabulary every later chapter depends on: radians, the unit circle, and periodicity — and shows a second, genuinely deeper wraparound problem, one that connects directly back to Numerical Methods & Floating-Point Computation's own representation-error material.

Degrees vs. Radians

A **radian** measures an angle by arc length: one radian is the angle subtended when the arc length equals the circle's own radius. A full circle is `2π` radians, which is why `π` radians equals exactly half a circle: `180°`.

VERIFIED DIRECTLY

`math.degrees(math.pi) = 180.0` exactly, and `math.radians(180) = 3.141592653589793` — Python's own stored approximation of `π`. Virtually every math library's trigonometric functions (`sin` , `cos` , `tan`) expect radians, not degrees — a genuinely common source of bugs when a value coming from user input or a design tool in degrees is passed straight into a trig function expecting radians.

The Unit Circle Definitions

For a point on a circle of radius `1` centered at the origin, at angle `θ` measured counterclockwise from the positive x-axis: `cos(θ)` is that point's x-coordinate, and `sin(θ)` is its y-coordinate. `tan(θ) = sin(θ)/cos(θ)` — the slope of the line from the origin to that point.

ANGLE	`cos(θ)` — X-COORD	`sin(θ)` — Y-COORD	QUADRANT BEHAVIOR
0°	1	0	Positive x-axis
90°	0	1	Positive y-axis
180°	−1	0	Negative x-axis
270°	0	−1	Negative y-axis

A Real Surprise: "Exact" Trig Identities Aren't Exactly True in Floating Point

The unit circle says `cos(90°)=0` and `sin(180°)=0` exactly. In floating point, these are only *approximately* true — for the same reason Numerical Methods & Floating-Point Computation's own Chapter 2 established: `π` itself has no exact binary representation, so `π/2` passed to `cos()` is never *quite* the true mathematical `π/2`.

VERIFIED DIRECTLY

`cos(math.pi/2) = 6.123233995736766×10⁻¹⁷` — not exactly `0`. `sin(math.pi) = 1.2246467991473532×10⁻¹⁶` — also not exactly `0`. Both errors are tiny, right at the scale of machine epsilon, and harmless for most purposes — but a piece of code that checks `if cos(angle) == 0:` to detect a right angle, following Numerical Methods & Floating-Point Computation's own Chapter 1 warning about exact equality checks, would **never** trigger, even at what's conceptually exactly `90°`.

A DRAMATIC CONSEQUENCE: TAN() NEAR ITS ASYMPTOTE

`tan(θ)` is mathematically undefined at `θ=90°`, since division by `cos(90°)=0` is undefined. In floating point, `tan(math.pi/2)` doesn't raise an error or return infinity — it returns `1.633123935319537×10¹⁶`, a huge but perfectly ordinary finite number, because `cos(π/2)` as actually computed is that tiny `6.12×10⁻¹⁷` value above, not true zero. This is exactly the "dividing by something close to zero" pattern Numerical Methods & Floating-Point Computation's own conditioning chapter flagged — code near a tangent asymptote needs an explicit check, since it will silently return a huge, misleading finite number rather than failing loudly.

Periodicity, and a Second, Deeper Wraparound Problem

`sin` and `cos` repeat every `2π`; `tan` repeats every `π` (since flipping both `sin` and `cos` in sign leaves their ratio unchanged). This means a game object's rotation angle can, in principle, be represented by infinitely many equivalent values — `θ`, `θ+2π`, `θ+4π`, and so on. In practice, letting an accumulating angle grow without ever wrapping it back into a bounded range is a real, measurable mistake.

VERIFIED DIRECTLY — 2,000,000 FRAMES OF UNWRAPPED ROTATION VS. WRAPPED ROTATION

Simulating a spinning object that adds a small fixed increment to its rotation angle every frame, for 2,000,000 frames, without ever wrapping: the final angle reaches `≈27,400` radians. Computing `sin()` directly on that huge, never-wrapped value gives a relative error of `≈3.15×10⁻⁷` against a high-precision reference. Doing the mathematically identical accumulation but wrapping the angle back into `[0, 2π)` **every single frame** instead gives a relative error of just `≈4.54×10⁻¹¹` — **nearly four orders of magnitude more accurate**, for the exact same sequence of increments.

WHY THIS HAPPENS — A DIRECT CALLBACK TO NUMERICAL METHODS & FLOATING-POINT COMPUTATION

A double's precision is *relative* to its own magnitude (Numerical Methods & Floating-Point Computation Chapter 2's own sliding exponent). An angle value near `27,400` has far coarser spacing between representable values than an

angle value kept near `2π≈6.28` — every increment added to the large unwrapped value loses more precision to rounding than the same increment added to a small, regularly-wrapped one. Regularly wrapping an accumulating angle isn't just tidier code — it's a genuine, measurable numerical-accuracy improvement, for free.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
`cos(π/2)` and `sin(π)` aren't exactly zero	A concrete reason Chapter 5's rotation-matrix code needs tolerance checks, not exact-equality checks, exactly as Numerical Methods & Floating-Point Computation Chapter 1 warned generally
An unwrapped accumulating angle loses real precision	Directly foreshadows Chapter 6's gimbal lock and Chapter 7's quaternions, both of which exist partly to keep repeated rotation accumulation numerically well-behaved
The unit circle's `(cos θ, sin θ)` point definition	The literal basis for every rotation matrix built in Chapter 5

Hands-On Exercises

EXERCISE 1

A design tool exports a rotation of `45` degrees, but a rendering function expects radians and receives the raw value `45` unconverted. Using this chapter's own degree/radian conversion, compute what angle (in degrees) the renderer will actually display, and explain why the bug might not be obvious from a quick glance at the number `45`.

View solution

EXERCISE 2

Using this chapter's own verified `cos(π/2)` and `tan(π/2)` results, explain why a physics engine checking `if abs(cos(angle)) < 1e-9:` to detect "this object is at a right angle" is a better design than checking `if cos(angle) == 0:`, and why a physics engine should specifically guard against dividing by `cos(angle)` near that same angle.

View solution

EXERCISE 3

Using this chapter's own verified 2,000,000-frame experiment, explain in your own words why wrapping an accumulating angle every frame is not just a stylistic preference but a genuine numerical-accuracy improvement — your answer should reference why a large angle value has less usable precision than a small one.

View solution

CHAPTER 2 QUICK REFERENCE

- **Radians:** a full circle is `2π` radians; `π` radians `=180°` — verified exactly via `math.degrees(math.pi)=180.0`
- **Unit circle:** `cos(θ)` = x-coordinate, `sin(θ)` = y-coordinate, `tan(θ)=sin(θ)/cos(θ)`, for a point at angle `θ` on a radius-1 circle
- Verified: `cos(π/2)≈6.12×10⁻¹⁷` and `sin(π)≈1.22×10⁻¹⁶` — not exactly zero, because `π` itself has no exact floating-point representation; `tan(π/2)` returns a huge but finite number (`≈1.63×10¹⁶`) rather than failing
- **Periodicity:** `sin` / `cos` repeat every `2π`; `tan` repeats every `π`
- Verified: an angle accumulated over 2,000,000 frames without wrapping has `≈3.15×10⁻⁷` relative error in its `sin()`; wrapping every frame reduces that to `≈4.54×10⁻¹¹` — nearly 4 orders of magnitude better, for free
- **Next chapter:** Triangles — the Law of Sines and the Law of Cosines, the practical toolkit for solving a triangle from partial information

CHAPTER 3 OF 10

Triangles: The Law of Sines, the Law of Cosines & Practical Trigonometry

Geometry & Trigonometry

Chapter 3 · Triangles: The Law of Sines, the Law of Cosines & Practical Trigonometry

Chapter 2 built the unit circle's vocabulary. This chapter puts it to work solving triangles from partial information — the real technique behind surveying, GPS-style triangulation, and any code that needs to turn a couple of measured angles and distances into a full picture of where something is.

The Law of Sines

For any triangle with sides `a, b, c` opposite angles `A, B, C` respectively: `a/sin(A) = b/sin(B) = c/sin(C)`. Given one full side-angle pair and one more piece of information, every other side and angle can be recovered.

The Law of Cosines

A generalization of the Pythagorean theorem to any triangle, not just right triangles: `c² = a² + b² − 2ab·cos(C)`. When `C=90°`, `cos(C)=0` and this collapses back to the familiar `c²=a²+b²`.

A Worked Triangulation Example

A surveyor wants the distance to a distant tower `T`, without being able to measure it directly. They walk a known baseline `AB=100m`, and measure the angle to the tower from each end: `60°` at `A`, `70°` at `B`.

VERIFIED DIRECTLY — SOLVING THE TRIANGLE WITH THE LAW OF SINES

The third angle is immediate: `C = 180° − 60° − 70° = 50°`. Applying the Law of Sines, `AT = AB·sin(B)/sin(T) = 100·sin(70°)/sin(50°) ≈ 122.67m`, and `BT = AB·sin(A)/sin(T) = 100·sin(60°)/sin(50°) ≈ 113.05m` — the tower's distance from each end of the baseline, computed without ever measuring it directly.

VERIFIED DIRECTLY — THE LAW OF COSINES AS AN INDEPENDENT CROSS-CHECK

Plugging the two just-computed distances back into the Law of Cosines, `AB = √(AT² + BT² − 2·AT·BT·cos(T))`, recovers `100.00000000000001` — the original baseline, to floating-point precision. The two laws aren't independent

facts to memorize separately; they're two consistent views of the same triangle, and cross-checking one against the other is a genuinely useful way to catch a measurement or calculation error before trusting the result.

Practical Trigonometry: The SSA "Ambiguous Case"

Not every combination of known sides and angles determines a triangle uniquely. Given two sides and a non-included angle (**S**ide-**S**ide-**A**ngle, where the angle isn't between the two given sides), the Law of Sines can produce **zero, one, or two** valid triangles from the exact same input — a real, well-known source of bugs in code that assumes a single formula always gives "the" answer.

VERIFIED DIRECTLY — THREE GENUINELY DIFFERENT OUTCOMES FROM THE SAME SETUP

Given side `b=10` and angle `A=40°`, varying only side `a`:

GIVEN SIDE `A`	`B·SIN(A)/A`	VERIFIED OUTCOME
`a=3`	`2.14` (>1)	**No valid triangle** — the required `sin(B)` would exceed `1`, an impossible value
`a=7`	`0.918`	**Two valid triangles**: `B≈66.67°` (giving side `c≈10.43`) or `B≈113.33°` (giving side `c≈4.89`) — both satisfy the same given `a`, `b`, and `A`
`a=15`	`0.428`	**Exactly one valid triangle**: the second candidate angle would push the angle sum past `180°`, so only `B≈25.37°` is geometrically possible

WHY THIS IS A REAL BUG SOURCE, NOT JUST A TEXTBOOK CURIOSITY

Code that computes `B = asin(b*sin(A)/a)` and stops there silently picks only *one* of the two mathematically valid answers whenever two exist (per the `a=7` row above), and crashes on a domain error whenever none exist (per the `a=3` row) without necessarily explaining why. A triangulation or navigation system relying on SSA-style measurements needs to explicitly check `b·sin(A)/a` against `1` first, and consider *both* `asin(x)` and `π−asin(x)` as candidates whenever a solution exists — exactly what the verified table above does.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
The Law of Sines/Cosines cross-check recovering the exact baseline	The same consistency-checking instinct Chapter 9 applies to intersection tests — verify a geometric result against an independent second computation
The SSA ambiguous case's `asin` / `π−asin` branch choice	Directly foreshadows Chapter 6's gimbal lock, where a similar loss of uniqueness (many different angle combinations producing the same orientation) causes real problems

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Solving a triangle from partial angle/distance measurements	The same underlying technique used by Chapter 9's own ray-based intersection tests

Hands-On Exercises

EXERCISE 1

Using this chapter's own Law of Sines formula, a surveyor measures a baseline of `200m` and angles of `50°` and `65°` at each end toward a landmark. Compute the distance from each end of the baseline to the landmark.

View solution

EXERCISE 2

Using this chapter's own verified SSA table, explain in your own words why `a=3` (with `b=10`, `A=40°`) produces no valid triangle at all, connecting your answer to what the value `b·sin(A)` geometrically represents.

View solution

EXERCISE 3

A GPS-style positioning system uses SSA-style triangulation (two known distances and one measured angle) to compute a device's position, and always takes the first (`asin`) solution without checking for a second one. Using this chapter's own `a=7` verified example, explain what could go wrong with this design, and under what circumstances the bug would actually manifest.

View solution

CHAPTER 3 QUICK REFERENCE

- **Law of Sines:** `a/sin(A) = b/sin(B) = c/sin(C)`
- **Law of Cosines:** `c² = a²+b²-2ab·cos(C)` — generalizes the Pythagorean theorem to any triangle
- Verified: a baseline-and-two-angles triangulation (surveying a tower) solved via the Law of Sines, then independently cross-checked via the Law of Cosines, recovering the original baseline to floating-point precision
- **SSA "ambiguous case":** given two sides and a non-included angle, verified all three real outcomes — zero valid triangles (`b·sin(A)/a > 1`), two valid triangles (an "ambiguous" middle range), or exactly one
- Code solving SSA-style problems must check `b·sin(A)/a` against `1` first, and consider both `asin(x)` and `π−asin(x)` as candidate answers

- **Next chapter:** Vectors and dot/cross products in geometric context — reusing Linear Algebra Fundamentals' own material, now applied to angles, projections, and surface normals

CHAPTER 4 OF 10

Vectors & Dot/Cross Products in Geometric Context

Geometry & Trigonometry

Chapter 4 · Vectors & Dot/Cross Products in Geometric Context

Linear Algebra Fundamentals defined the dot and cross products algebraically. This chapter puts them to work on genuinely geometric problems: decomposing a vector into useful directional components, finding the direction a surface faces, and computing how brightly a light illuminates it — three of the most common building blocks in real graphics and game code.

Vector Projection: Splitting a Vector Into Two Useful Parts

Given a vector `a` and a direction `b`, the **projection** of `a` onto `b` is the component of `a` that points along `b`: `proj_b(a) = (a·b / b·b) · b`. Whatever's left over, `a − proj_b(a)`, is the component of `a` perpendicular to `b`.

VERIFIED DIRECTLY — A SELF-CHECKING DECOMPOSITION

For `a=(5,3)` projected onto `b=(2,1)`: `proj_b(a) = (5.2, 2.6)`, and the perpendicular remainder is `(-0.2, 0.4)`. Two independent checks confirm this is correct: `proj + perp` recovers `(5.0, 3.0)` — exactly the original vector `a` — and the dot product of the perpendicular component with `b` is `≈-4.44×10⁻¹⁶`, effectively zero, confirming the two really are perpendicular.

This decomposition is exactly how a game physics engine splits gravity into "the component pulling a character down a slope" (the projection onto the slope's own direction) and "the component pressing into the slope" (the perpendicular remainder) — two physically meaningful quantities recovered from one vector and one direction.

Surface Normals via the Cross Product

Every triangle in a 3D mesh has a direction it "faces" — its **normal** vector, perpendicular to the triangle's own surface. Given two of the triangle's edges as vectors, the cross product gives exactly that: `normal = edge1 × edge2`, normalized to unit length.

VERIFIED DIRECTLY — A REAL 3D TRIANGLE'S NORMAL, CHECKED FOR CORRECTNESS

For a triangle with vertices `A=(0,0,0)`, `B=(2,0,0)`, `C=(0,3,1)`: the raw cross product of `edge1=B−A` and `edge2=C−A` gives `(0, −2, 6)`, normalizing to the unit vector `(0, −0.3162, 0.9487)`, confirmed to have length

`0.9999999999999999≈1`. Two independent checks confirm it's genuinely perpendicular to the triangle: the dot product of the (unnormalized) normal with *both* edges comes out to exactly `0`.

WHY THE ORDER OF THE CROSS PRODUCT MATTERS

`edge1 × edge2` and `edge2 × edge1` point in exactly opposite directions (the cross product anticommutes). Which order a mesh format uses determines whether a triangle's normal faces "outward" or "inward" — get it backward, and every triangle in a model appears to face the wrong way, a real and common source of "inside-out" looking 3D models.

Lighting: Why the Dot Product Needs a Clamp

The simplest realistic lighting model, **Lambertian (diffuse) lighting**, computes a surface's brightness as `max(0, normal · light_direction)` — the dot product between the surface normal and the direction toward the light.

VERIFIED DIRECTLY — LIGHT IN FRONT VS. LIGHT BEHIND THE SAME SURFACE

Using the triangle normal computed above: with a light positioned generally in front of the surface, `normal · light_direction ≈ 0.6069` — a sensible, positive brightness. With the exact same light instead positioned *behind* the surface (the mirror-image direction), the dot product comes out to `≈-0.6069` — a **negative** brightness, which is physically meaningless (a surface can't emit negative light). The `max(0, ...)` clamp exists exactly to catch this: it correctly reduces the negative result to `0`, meaning "this surface receives no light from this direction at all."

A REAL, COMMON BUG: FORGETTING THE CLAMP

Skipping `max(0, ...)` and using the raw dot product directly is a genuinely common graphics bug — a negative brightness value fed straight into a color channel either gets silently clamped somewhere else in the rendering pipeline (masking the bug) or produces visibly wrong, "negative-lit" dark patches on surfaces facing away from every light source. This is exactly the same defensive-clamping instinct as checking a value's valid range before using it — familiar territory from Numerical Methods & Floating-Point Computation's own emphasis on not trusting a raw computed value without checking it makes sense.

Bonus: Which Side of a Line Is a Point On?

The 2D cross product's *sign* (not magnitude) answers a genuinely useful question directly: for a line from `P1` to `P2`, and a point `Q`, the sign of `cross(P2-P1, Q-P1)` tells you which side of the line `Q` is on.

VERIFIED DIRECTLY

For the line from `(0,0)` to `(4,0)` (the x-axis): a point `(2,3)` above the line gives a cross-product value of `+12`; a point `(2,-3)` below the line gives `-12` — same magnitude, opposite sign, exactly tracking which side each point falls on.

Chapter 9 builds this exact sign test into a full point-in-polygon and line-intersection toolkit.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Vector projection splits a vector into parallel/perpendicular parts	The literal mechanism behind resolving a rotation into axis components in Chapter 5-6
Cross product order determines normal direction	The same ordering sensitivity reappears in Chapter 5's rotation composition, where order changes the result
The 2D cross-product sign test for "which side"	Reused directly as the core mechanism behind Chapter 9's line-intersection and point-in-polygon tests

Hands-On Exercises

EXERCISE 1

Using this chapter's own projection formula, decompose `a=(6,2)` onto the direction `b=(1,3)`. Compute the parallel and perpendicular components, and verify both that they sum back to `a` and that the perpendicular component is orthogonal to `b`.

View solution

EXERCISE 2

A 3D model appears "inside-out" after being loaded — every surface that should be visible from outside is instead invisible (culled), and vice versa. Using this chapter's own explanation of cross-product order and normals, explain the most likely cause and how you would confirm it.

View solution

EXERCISE 3

Using this chapter's own verified lighting example, explain why a renderer that skips the `max(0, ...)` clamp might still "look correct" in many scenes during testing, and describe a specific scene setup where the bug would become clearly visible.

View solution

CHAPTER 4 QUICK REFERENCE

- **Vector projection:** `proj_b(a) = (a·b/b·b)·b`; the perpendicular remainder is `a - proj_b(a)` — verified self-checking: the two parts sum back to `a`, and are mutually orthogonal (dot product `≈0`)
- **Surface normal:** `edge1 × edge2`, normalized — verified genuinely perpendicular to both triangle edges (dot products exactly `0`) and unit length
- Cross-product order matters: `edge1×edge2 ≠ edge2×edge1` — reversing it flips which way a normal (and therefore a whole model's visible surfaces) faces
- **Lambertian lighting:** `max(0, normal·light_direction)` — verified a light positioned behind a surface gives a negative dot product (`≈-0.6069`), which the clamp correctly reduces to `0`
- The 2D cross product's *sign* tells you which side of a line a point is on — verified `+12` above vs. `-12` below the same line
- **Next chapter:** 2D rotations and rotation matrices — a deeper pass on Linear Algebra Fundamentals' own brief rotation-matrix introduction

CHAPTER 5 OF 10

2D Rotations & Rotation Matrices

Geometry & Trigonometry

Chapter 5 · 2D Rotations & Rotation Matrices

Linear Algebra Fundamentals introduced the 2D rotation matrix briefly, as one example transformation among several. This chapter gives it the deeper, dedicated treatment it needs: composing rotations, rotating about a point other than the origin, why the order transforms are applied in genuinely changes the result, and a real, verified reason repeated rotation composition needs care — directly setting up Chapters 6 and 7's own motivation for quaternions.

The 2D Rotation Matrix, Built From the Unit Circle

Chapter 2 defined `(cos θ, sin θ)` as the point reached by rotating `(1,0)` by angle `θ`. The rotation matrix is exactly that idea generalized to rotate *any* point: `R(θ) = [[cos θ, −sin θ], [sin θ, cos θ]]`.

Composing Rotations

Applying `R(θ₁)` and then `R(θ₂)` should be the same as applying one combined rotation, `R(θ₁+θ₂)` — and matrix multiplication makes that literal: `R(θ₂)·R(θ₁) = R(θ₁+θ₂)`.

VERIFIED DIRECTLY

`R(30°)·R(45°)`, computed as an actual matrix product, gives `[[0.25881904510252085, -0.9659258262890683], [0.9659258262890683, 0.25881904510252085]]`. Computing `R(75°)` directly gives `[[0.25881904510252074, -0.9659258262890683], [0.9659258262890683, 0.25881904510252074]]` — matching to within `≈10⁻¹⁶`, exactly the trigonometric angle-addition identities in matrix form.

Rotating About an Arbitrary Pivot

The rotation matrix alone always rotates around the origin. To rotate around a different pivot point `C`: translate the point so `C` becomes the origin, rotate, then translate back — `P' = C + R(θ)(P−C)`.

VERIFIED DIRECTLY — MATCHING A HAND CALCULATION

Rotating `P=(5,5)` by `90°` around pivot `C=(2,2)`: subtracting the pivot gives `(3,3)`; rotating `90°` maps `(x,y)→(−y,x)`, giving `(−3,3)`; adding the pivot back gives `P' = (−1, 5)` — confirmed exactly by the matrix

computation.

Why Order Matters: Rotate-Then-Translate vs. Translate-Then-Rotate

Combining a rotation and a translation is **not commutative** — doing them in the opposite order produces a genuinely different final position, not just a different intermediate path to the same place.

VERIFIED DIRECTLY — THE SAME TWO OPERATIONS, DRAMATICALLY DIFFERENT RESULTS

Starting from `(1,0)`, rotating `90°` then translating by `(5,0)` gives `(5.0, 1.0)`. Translating by `(5,0)` *first*, then rotating `90°`, gives `(≈0, 6.0)` — a completely different final point, from the identical rotation and the identical translation, just applied in the opposite order.

WHY THIS IS A REAL, COMMON BUG

This is exactly the mechanism behind "my object orbits around the wrong point" bugs in game engines: an object rotated after being moved away from the origin rotates around the *origin*, not around its own current position, producing a wide, unintended orbiting motion instead of a stationary spin. The fix is precisely this chapter's own arbitrary-pivot formula — rotate around the object's own position as the pivot, not the world origin.

A Real Reason Rotation Matrices Need Care: Accumulated Drift

Chapter 2 verified that accumulating a rotation *angle* without wrapping it loses precision. Composing rotation *matrices* by repeated multiplication has an analogous, independently verifiable problem.

VERIFIED DIRECTLY — 200,000 COMPOSED ROTATIONS VS. ONE DIRECT COMPUTATION

Composing a small rotation matrix (`0.0003` radians) with itself `200,000` times via repeated matrix multiplication (total angle: `60` radians) gives a determinant of `1.0000000000188298`. A true rotation matrix always has determinant **exactly** `1` — computing the equivalent single rotation matrix directly, `R(60 rad)`, gives a determinant of `0.9999999999999999`, essentially exact. The composed matrix has visibly drifted away from being a genuine rotation at all — its rows and columns are no longer quite perpendicular unit vectors — with a maximum per-entry difference of `≈8.97×10⁻¹²` compared to the direct computation.

WHY THIS MATTERS, AND WHAT REAL ENGINES DO ABOUT IT

An object whose orientation is updated every frame by multiplying its current rotation matrix by a small incremental rotation — a completely natural way to implement continuous spinning — accumulates exactly this drift over time, the same way Chapter 2's unwrapped angle accumulator lost precision. Left unchecked, the matrix can gradually stop being a pure rotation at all, distorting the object it represents (subtly scaling or shearing it, not just rotating it). Real engines periodically **re-orthonormalize** the matrix to correct this drift — and, as Chapter 7 covers, switching to quaternions sidesteps a large part of this problem in the first place.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Composing rotations via matrix multiplication	Directly extends to 3D in Chapter 6 — with a genuinely new complication (order-dependence around different axes) that 2D rotation alone can't show
Arbitrary-pivot rotation via translate-rotate-translate-back	The same three-step pattern reappears in Chapter 8's coordinate-system transformations
Accumulated matrix-composition drift, verified over 200,000 steps	The central, motivating problem Chapter 7's quaternions are specifically designed to reduce

Hands-On Exercises

EXERCISE 1

Using this chapter's own composition rule, verify by hand (using the angle-addition trigonometric identities `cos(a+b)=cos a cos b − sin a sin b` and `sin(a+b)=sin a cos b + cos a sin b`) that `R(30°)·R(45°)`'s top-left entry should equal `cos(75°)`, and confirm it matches this chapter's own verified numeric result.

View solution

EXERCISE 2

A game character standing at position `(10, 0)` needs to spin in place by `180°`. Using this chapter's own arbitrary-pivot formula, compute the character's new position if the code incorrectly rotates around the world origin `(0,0)` instead of the character's own position, and explain what the player would actually observe.

View solution

EXERCISE 3

Using this chapter's own verified 200,000-step drift experiment, explain why a game engine that updates an object's rotation by matrix-multiplying a small incremental rotation onto it every frame, for an object that spins continuously for a very long play session, could eventually cause visible problems beyond just "the angle is slightly off" — what specifically would start to look wrong?

View solution

CHAPTER 5 QUICK REFERENCE

- **2D rotation matrix:** `R(θ) = [[cos θ, -sin θ],[sin θ, cos θ]]`, built directly from Chapter 2's unit-circle point definition
- Verified: `R(30°)·R(45°)` matches `R(75°)` to within `≈10⁻¹⁶` — matrix composition is angle addition
- **Arbitrary pivot:** `P' = C + R(θ)(P−C)` — verified matching a hand calculation exactly
- Verified: rotate-then-translate and translate-then-rotate give genuinely different results (`(5,1)` vs. `(≈0,6)`) from identical individual operations — the real mechanism behind "orbiting around the wrong pivot" bugs
- Verified: 200,000 composed small rotations drift to determinant `1.0000000000188298` (should be exactly `1`) vs. a fresh direct computation's `0.9999999999999999` — real, measurable degradation from repeated matrix composition
- **Next chapter:** 3D rotations, Euler angles, and gimbal lock — where composing rotations gets a genuinely new complication

CHAPTER 6 OF 10

3D Rotations: Euler Angles & Gimbal Lock

Geometry & Trigonometry

Chapter 6 · 3D Rotations: Euler Angles & Gimbal Lock

Chapter 5 built 2D rotation to real depth — composition, arbitrary pivots, and a verified drift problem. The natural next step, extending rotation matrices to three dimensions and describing an orientation as three sequential rotations, seems straightforward. It has a real, mathematically inevitable failure mode: **gimbal lock**, verified concretely in this chapter, not just described.

3D Rotation Matrices: One Per Axis

Where 2D rotation had one matrix, 3D rotation needs three — one rotation around each axis:

```
Rx(θ) = [[1,      0,       0],
         [0, cos θ, -sin θ],
         [0, sin θ,  cos θ]]

Ry(θ) = [[ cos θ, 0, sin θ],
         [     0, 1,     0],
         [-sin θ, 0, cos θ]]

Rz(θ) = [[cos θ, -sin θ, 0],
         [sin θ,  cos θ, 0],
         [    0,      0, 1]]
```

A New Problem 2D Never Had: Rotations Around Different Axes Don't Commute

Chapter 5 showed that combining a rotation with a *translation* doesn't commute. In 3D, even combining two *rotations* around different axes — with no translation at all — doesn't commute either.

VERIFIED DIRECTLY — THE SAME TWO 90° ROTATIONS, APPLIED IN OPPOSITE ORDERS

Applying `Rx(90°)` then `Ry(90°)` to the point `(0,0,1)` gives `(0, −1, 0)`. Applying the exact same two rotations in the opposite order — `Ry(90°)` then `Rx(90°)` — gives `(1, 0, 0)`. Two completely different points on the unit sphere, from the identical pair of 90° rotations, differing only in which was applied first.

Euler Angles: Describing an Orientation as Three Sequential Rotations

An **Euler angle** representation describes any 3D orientation as three rotations applied in a chosen order around a chosen set of axes — commonly **yaw** (around `z`), **pitch** (around `y`), and **roll** (around `x`), combined as `R = Rz(yaw)·Ry(pitch)·Rx(roll)`. It's intuitive — three familiar, independent-feeling knobs — and it's exactly how aircraft and camera orientation are usually described in plain language.

Gimbal Lock: A Real, Verified Loss of a Degree of Freedom

"Independent-feeling" is doing a lot of work in that sentence above. At one specific pitch value, `90°`, yaw and roll stop being independent at all.

VERIFIED DIRECTLY — FOUR GENUINELY DIFFERENT YAW/ROLL PAIRS, ONE IDENTICAL ROTATION MATRIX

Computing the full orientation matrix `R = Rz(yaw)·Ry(90°)·Rx(roll)` for four different `(yaw, roll)` pairs that all share the same difference, `yaw − roll = 20°` — `(30°,10°)`, `(40°,20°)`, `(25°,5°)`, and `(100°,80°)` — produces **exactly the same rotation matrix** in all four cases, matching to floating-point precision: `[[0, -0.342, 0.940], [0, 0.940, 0.342], [-1, 0, 0]]`.

WHY THIS PROVES A GENUINE, PERMANENT LOSS — NOT JUST "HARD TO CONTROL"

This isn't a coincidence or a rounding artifact — it can be derived algebraically: at `pitch=90°`, `Rz(yaw)·Ry(90°)·Rx(roll)` simplifies to a matrix that depends *only* on `yaw − roll`, never on yaw and roll individually. Two full turning knobs that felt independent everywhere else have collapsed into **one** effective parameter. An animation or flight-control system trying to adjust yaw and roll independently at this exact orientation would find that changing either one alone produces the identical visible rotation as changing the other — one entire degree of freedom of control has genuinely vanished, not merely become awkward.

"Gimbal lock" gets its name from a physical mechanical gimbal — a set of nested rotating rings, historically used in navigation instruments and spacecraft attitude systems, that suffers the exact same failure for the exact same geometric reason when two of its rings' axes become aligned.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
3D rotation order-dependence, verified with two 90° rotations	A direct extension of Chapter 5's own 2D order-dependence finding into a genuinely new dimension of complexity
Gimbal lock verified as an exact, derivable loss of a degree of freedom	The central, motivating problem Chapter 7's quaternions are specifically built to avoid — not by making gimbal lock "less bad," but by using a representation that structurally can't exhibit it

THIS CHAPTER'S FINDING	WHAT IT SETS UP
The four-parameter matrix (3 Euler angles chosen from many possible axis orders)	Chapter 8's coordinate-system transformations, which build on the same rotation-matrix machinery for full 3D scene transforms

Hands-On Exercises

EXERCISE 1

Using this chapter's own `Rx` and `Ry` matrices, apply `Rx(90°)` to the point `(0,1,0)`, then apply `Ry(90°)` to the result. Separately, apply `Ry(90°)` to `(0,1,0)` first, then apply `Rx(90°)` to that result. Confirm the two final points are different.

View solution

EXERCISE 2

Using this chapter's own verified gimbal-lock finding (that the resulting matrix at `pitch=90°` depends only on `yaw − roll`), predict without recomputing whether `(yaw=60°, roll=40°)` and `(yaw=15°, roll=−5°)` would produce the same rotation matrix at `pitch=90°`, and explain your reasoning.

View solution

EXERCISE 3

A flight simulator's camera uses yaw/pitch/roll Euler angles and lets the player independently control yaw and roll with two separate joystick axes. Using this chapter's own gimbal-lock finding, describe specifically what the player would experience if the camera's pitch reached exactly 90° (looking straight up) while they tried to use both control axes.

View solution

CHAPTER 6 QUICK REFERENCE

- **3D rotation matrices:** `Rx`, `Ry`, `Rz`, one per axis, each a direct extension of Chapter 5's 2D matrix
- Verified: rotating `(0,0,1)` by `Rx(90°)` then `Ry(90°)` gives `(0,−1,0)`; the opposite order gives `(1,0,0)` — 3D rotations don't commute even without translation
- **Euler angles:** yaw/pitch/roll as three sequential axis rotations, `R=Rz(yaw)·Ry(pitch)·Rx(roll)`
- **Gimbal lock, verified:** at `pitch=90°`, four genuinely different `(yaw,roll)` pairs sharing the same `yaw−roll` value produce *exactly* the same rotation matrix — a real, derivable, permanent loss of one degree of freedom, not just a control inconvenience

- **Next chapter:** Quaternions — the standard practical fix, using a representation that structurally can't exhibit gimbal lock

CHAPTER 7 OF 10

Quaternions

Geometry & Trigonometry

Chapter 7 · Quaternions

Chapter 6 verified a real, permanent loss of a degree of freedom whenever an orientation is built from three sequential Euler-angle rotations at `pitch=90°`. Quaternions are the standard practical fix used throughout real games, robotics, and computer-vision code — and this chapter treats them as a genuinely usable tool built from four numbers you can compute and check by hand, not a black box to be imported and trusted blindly.

What a Quaternion Actually Is

A **unit quaternion** representing a rotation of angle `θ` around a unit axis `(a_x,a_y,a_z)` is four numbers: `q = (cos(θ/2), a_x·sin(θ/2), a_y·sin(θ/2), a_z·sin(θ/2))`. The first component is often called `w`; the other three, `(x,y,z)`, encode the rotation axis scaled by `sin(θ/2)`. Notice the **half-angle** — a genuinely easy-to-forget detail that trips up a first implementation.

VERIFIED DIRECTLY — A QUATERNION CORRECTLY ROTATING A VECTOR

The quaternion for a `90°` rotation around the z-axis: `q = (0.7071, 0, 0, 0.7071)` — cosine and sine of the *half*-angle, `45°`. Rotating the vector `(1,0,0)` using the standard formula `v' = q·v·q⁻¹` (treating `v` as a "pure" quaternion `(0,v_x,v_y,v_z)`) gives exactly `(0, 1, 0)` — matching the `Rz(90°)` matrix rotation from Chapter 5 precisely.

Composing Rotations: Quaternion Multiplication Reproduces the Matrix Result Exactly

Quaternions compose the same way rotation matrices do: multiplying two quaternions gives the quaternion for the combined rotation. The real test is whether this actually reproduces Chapter 6's own matrix results, number for number.

VERIFIED DIRECTLY — REPRODUCING CHAPTER 6'S OWN NON-COMMUTATIVITY FINDING EXACTLY

Composing the quaternions for `Rx(90°)` and `Ry(90°)`, then rotating `(0,0,1)`: applying `Rx` then `Ry` gives `(0, -1, 0)`; applying `Ry` then `Rx` gives `(1, 0, 0)` — **exactly** Chapter 6's own verified matrix results, to the same

precision. Quaternions aren't a different kind of rotation from matrices; they're a different *encoding* of the identical underlying rotations, verified to agree completely.

What Quaternions Actually Fix — and an Honest Limit

It's tempting to say "quaternions eliminate gimbal lock." That's *almost* right, and the precise version matters.

VERIFIED DIRECTLY — COMPOSING ROTATIONS VIA QUATERNIONS NEVER ROUTES THROUGH A SINGULAR PARAMETERIZATION

Quaternion multiplication never decomposes an orientation into three sequential single-axis rotations the way Euler angles do — every orientation is one point on a smooth, unbroken 4D unit sphere, with no special "pitch=90°" coordinate patch where the representation itself becomes degenerate. Composing and — as covered briefly below — smoothly interpolating between orientations using quaternions stays numerically well-behaved *through* exactly the kind of orientation that broke Euler-angle composition in Chapter 6.

THE HONEST LIMIT: CONVERTING BACK TO EULER ANGLES DOESN'T ESCAPE THE UNDERLYING FACT

Building the quaternion for `(yaw=30°, pitch=90°, roll=10°)` and for `(yaw=40°, pitch=90°, roll=20°)` — the same `yaw-roll=20°` pair Chapter 6 verified collapses to one matrix — produces, verified directly, the **same quaternion** in both cases (matching to floating-point precision). Converting either one back into Euler angles for display recovers the identical `(yaw=180°, pitch=90°, roll=180°)` — the individual yaw and roll values are gone, exactly as they were for the matrix version. This isn't a shortcoming of quaternions specifically: at that exact orientation, the original `(yaw,roll)` distinction was never real information the orientation itself carried. Quaternions fix the *numerical behavior of composing and interpolating rotations*; they don't — and can't — restore information that a particular Euler-angle decomposition never uniquely had in the first place.

A Second Real Advantage: Cheap Renormalization

Chapter 5 verified that repeatedly composing rotation matrices drifts away from a true rotation, needing an involved re-orthonormalization to fix. A unit quaternion has exactly one constraint to maintain: its magnitude must stay `1`.

VERIFIED DIRECTLY — THE SAME KIND OF DRIFT, A MUCH CHEAPER FIX

Composing a small quaternion rotation with itself `200,000` times (mirroring Chapter 5's own matrix experiment): the resulting quaternion's magnitude drifts to `1.000000000009458`, instead of exactly `1`, with a maximum component difference of `≈9.34×10⁻¹²` from a fresh direct computation. Fixing this needs only **one** operation — dividing all four components by the quaternion's own magnitude — after which the difference from the direct computation shrinks to `≈1.88×10⁻¹⁴`, roughly **500× better**, from a single square root and four divisions. Correcting a drifted rotation matrix instead requires re-orthonormalizing an entire 3×3 matrix (typically via Gram-Schmidt across all three rows) — genuinely more arithmetic for a comparable fix.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Quaternion multiplication verified to exactly match matrix composition	Confirms Chapter 8's coordinate transformations can freely mix quaternion-based and matrix-based rotation representations, since they compute identical results
Quaternions stay numerically smooth through what would be a gimbal-lock orientation	The practical reason virtually every animation system interpolates camera/character orientation with quaternions (spherical linear interpolation, "SLERP") rather than interpolating Euler angles directly — a natural extension beyond this course's own scope
Cheap renormalization vs. expensive matrix re-orthonormalization	A concrete, quantified reason real engines default to quaternions for any orientation that's updated incrementally over many frames

Hands-On Exercises

EXERCISE 1

Using this chapter's own quaternion-construction formula, build the quaternion for a `180°` rotation around the y-axis, and use it to rotate the point `(0,0,1)`. Confirm your result matches what you'd expect from the `Ry(180°)` rotation matrix directly.

View solution

EXERCISE 2

A developer claims "since I switched my engine's orientation representation to quaternions, gimbal lock is completely impossible in my game now, even in the UI that displays the camera's yaw/pitch/roll to the player." Using this chapter's own verified findings, explain what part of this claim is correct and what part is not.

View solution

EXERCISE 3

Using this chapter's own verified drift-and-renormalization numbers, explain why a game engine might choose to renormalize an object's orientation quaternion every single frame, even though the drift after just one frame's worth of composition is far too small to be visible.

View solution

CHAPTER 7 QUICK REFERENCE

- **Quaternion from axis-angle:** `q = (cos(θ/2), a_x sin(θ/2), a_y sin(θ/2), a_z sin(θ/2))` — note the half-angle
- Verified: quaternion rotation of `(1,0,0)` by `90°` around `z` matches the `Rz(90°)` matrix result exactly, `(0,1,0)`
- Verified: quaternion multiplication exactly reproduces Chapter 6's own matrix non-commutativity results — `(0,-1,0)` and `(1,0,0)` for the two rotation orders
- Quaternions fix the *numerical behavior* of composing/interpolating rotations near a gimbal-lock orientation — verified, they don't restore Euler-angle information that was never uniquely recoverable there in the first place (both `(30°,10°)` and `(40°,20°)` yaw/roll pairs collapse to the identical quaternion)
- Verified: quaternion drift after 200,000 compositions (magnitude `1.000000000009458`) is fixed to `≈1.88×10⁻¹⁴` error by one cheap renormalization — versus a full matrix re-orthonormalization needed for the equivalent matrix drift
- **Next chapter:** Coordinate systems and transformations — world/local/camera/screen space, building on this chapter's own rotation machinery

CHAPTER 8 OF 10

Coordinate Systems & Transformations

Geometry & Trigonometry

Chapter 8 · Coordinate Systems & Transformations

Chapters 5-7 handled rotation in real depth, but a rotation matrix alone can't move something — Chapter 5 had to bolt translation on separately with a three-step "translate, rotate, translate back" pattern just to rotate around a non-origin pivot. This chapter reuses **homogeneous coordinates**, briefly introduced back in Linear Algebra Fundamentals Chapter 5, to fold rotation and translation into one matrix — and builds the full local/world/camera pipeline every real graphics or game engine runs every single frame.

Homogeneous Coordinates: One Matrix for Rotation + Translation

Adding a third coordinate, always `1`, to every 2D point turns translation into ordinary matrix multiplication: `T = [[cos θ, −sin θ, tₓ], [sin θ, cos θ, t_y], [0, 0, 1]]` applied to `(x, y, 1)` rotates *and* translates in a single matrix-vector product.

WHAT THIS ACTUALLY BUYS YOU, CONCRETELY

Chapter 5's arbitrary-pivot rotation needed three separate steps: subtract the pivot, rotate, add the pivot back. A single homogeneous transformation matrix folds all of that — and any further translations — into one matrix, which is also exactly why chaining multiple transformations (an object's own local transform, then its parent's transform, then the camera's transform) is just repeated matrix multiplication, no special-casing required.

Local, World, Camera, and Screen Space

SPACE	WHAT COORDINATES MEAN HERE
Local (object) space	Coordinates relative to the object's own center/origin — where its own vertices are authored, independent of where it's placed in the scene
World space	The scene's shared, global coordinate system — every object's local space is transformed into this one common space
Camera (view) space	World space re-expressed relative to the camera's own position and orientation — as if the camera sat at the origin looking down a fixed axis
Screen (clip) space	The final 2D pixel coordinates after projection — beyond this course's own scope, but built directly on everything above

Change of Basis: Getting From World Space Into Camera Space

The camera itself has a world position and orientation, exactly like any other object — described by its own `T_camera_to_world` matrix. Going the *other* direction, from world space into the camera's own frame of reference, needs the **inverse** of that matrix.

A GENUINE SHORTCUT: ROTATION MATRICES ARE ORTHOGONAL

Because a rotation matrix `R` satisfies `R⁻¹ = Rᵀ` (its own transpose — a direct consequence of its rows/columns being perpendicular unit vectors, per Chapter 4's own dot-product orthogonality checks), the inverse of a rotate-then-translate matrix doesn't need a full, general matrix inversion at all: `T⁻¹ = [[Rᵀ, −Rᵀt], [0, 1]]` — transpose the rotation part, and use it to un-translate.

A Full Verified Pipeline: Local → World → Camera

A "forward" marker point at local `(1, 0)` belongs to an object placed at world position `(5, 2)`, rotated `30°`. A camera sits at world position `(10, 10)`, rotated `−45°`.

VERIFIED DIRECTLY — STEP BY STEP, AND AS A SINGLE COMBINED MATRIX

Local → world: applying the object's transform to `(1,0,1)` gives `(5.866025403784438, 2.5, 1)` — matching the hand-check `(cos30°, sin30°) + (5,2)` exactly. **World → camera:** computing the camera's inverse transform via the orthogonality shortcut above, then applying it, gives `(2.380139..., −8.226462..., 1)`. Computing the *entire* local-to-camera transform as one combined matrix product first, then applying it directly to the original local point in a single step, gives **exactly the same result** — a maximum difference of `0.0` between the two computation paths.

VERIFIED DIRECTLY — THE INVERSE REALLY IS THE INVERSE

Multiplying the camera's own `camera→world` matrix by the computed `world→camera` matrix gives the identity matrix exactly: `[[1,0,0],[0,1,0],[0,0,1]]` (to the displayed precision) — confirming the change-of-basis matrix genuinely undoes the camera's own transform, not just approximately.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Rotation + translation as one homogeneous matrix	Directly generalizes to 3D with 4×4 matrices, using the exact same `Rx` / `Ry` / `Rz` building blocks from Chapter 6
The orthogonal-matrix inverse shortcut, `R⁻¹=Rᵀ`	A direct callback to Chapter 4's own dot-product orthogonality checks, now used as a genuine computational shortcut rather than just a verification tool

THIS CHAPTER'S FINDING	WHAT IT SETS UP
A consistent, verified coordinate pipeline	Chapter 9's intersection tests all assume every object being tested has already been placed into one shared, consistent coordinate space — exactly what this chapter builds

Hands-On Exercises

EXERCISE 1

Using this chapter's own homogeneous transformation matrix, build the local-to-world matrix for an object at world position `(3, 4)` rotated `90°`, and use it to find the world-space position of the local point `(2, 0)`.

View solution

EXERCISE 2

Using this chapter's own orthogonal-matrix inverse shortcut, explain why $R^{-1}=R^{T}$ is specifically true for a rotation matrix but would **not** be true for a general matrix that also scales an object (for example, one that stretches an object twice as wide as it is tall).

View solution

EXERCISE 3

A game engine transforms 10,000 object vertices from local space into camera space every frame. Using this chapter's own verified finding that computing a combined transform once and applying it directly gives exactly the same result as applying each step separately, explain why precomputing the single combined local-to-camera matrix once per object (rather than transforming each vertex through local→world→camera as three separate matrix multiplications) is a meaningful performance improvement.

View solution

CHAPTER 8 QUICK REFERENCE

- **Homogeneous coordinates:** a 3rd coordinate (`=1`) lets rotation and translation combine into one matrix, applied via ordinary matrix-vector multiplication
- **Pipeline spaces:** local (object) → world (shared scene) → camera (relative to the viewer) → screen (final pixels, beyond this course's scope)
- **Change of basis:** going from world space into camera space needs the camera's own transform *inverted* — and because rotation matrices are orthogonal, $R^{-1}=R^{T}$ avoids a full matrix inversion

- Verified: a full local→world→camera pipeline computed step by step matches a single precomputed combined matrix exactly (`0.0` difference), and the computed inverse correctly undoes the camera's transform (product = exact identity)
- **Next chapter:** Geometric primitives and intersection tests — line-line, ray-sphere, ray-plane, and point-in-polygon, all assuming objects already share one consistent coordinate space

CHAPTER 9 OF 10

Geometric Primitives & Intersection Tests

Geometry & Trigonometry

Chapter 9 · Geometric Primitives & Intersection Tests

Chapter 4 introduced the cross-product's sign as a "which side" test. Chapter 8 built a consistent coordinate pipeline every object shares. This chapter assembles the actual toolkit that answers "do these two things touch?" — the real question behind collision detection, mouse-picking, and ray tracing.

Line-Line Intersection

For two lines through `(P1,P2)` and `(P3,P4)`, the standard formula finds the intersection directly, with the denominator itself signaling the degenerate case:

VERIFIED DIRECTLY

Lines through `(0,0)-(4,4)` and `(0,4)-(4,0)` — an X shape — intersect at exactly `(2, 2)`, the visually obvious crossing point. Lines through `(0,0)-(4,4)` and `(1,0)-(5,4)` — two parallel diagonals — give a denominator of exactly `0`, correctly signaling "no intersection" rather than a division error, since parallel lines never cross.

Ray-Sphere Intersection: A Real Application of Numerical Methods & Floating-Point Computation's Own Quadratic Formula

Substituting a ray's parametric equation `O+tD` into a sphere's equation `|P−C|²=r²` produces a genuine quadratic in `t`: `at²+bt+c=0`, where `a=D·D`, `b=2D·(O−C)`, and `c=(O−C)·(O−C)−r²`. This is *exactly* the quadratic formula Numerical Methods & Floating-Point Computation Chapter 4 already verified can be numerically dangerous — and ray-sphere intersection is one of the most common places that danger shows up in real code.

VERIFIED DIRECTLY — A REAL RAY-SPHERE SETUP REPRODUCING THAT CHAPTER'S EXACT NUMBERS

Ray origin `O=(50000,0,0)`, direction `D=(−1,0,0)`, sphere centered at `C=(0,0,0)` with squared radius `r²=2,499,999,999` (radius `≈49999.99999`) — a physically ordinary setup: a ray starting far from a large sphere, aimed at it. The resulting quadratic coefficients come out to exactly `a=1`, `b=−100000`, `c=1` — **the identical coefficients** Numerical Methods & Floating-Point Computation Chapter 4 already solved. The naive `(−b±√disc)/2a` formula's near intersection point has a relative error of `≈3.38×10⁻⁷`; the stable ("Citardauq") reformulation gives

`≈1.14×10⁻¹⁶` — the exact same nine-orders-of-magnitude improvement already verified there, now confirmed in a genuine geometric context rather than an abstract equation.

WHY THIS MATTERS SPECIFICALLY FOR RAY TRACING

The *near* intersection point — exactly the one this chapter just showed is at risk — is the one that actually matters for rendering: it's the first surface the ray hits, the one that determines what color the pixel is. A ray tracer using the naive formula doesn't fail loudly; it just renders a very slightly wrong hit point on that first surface, most visible on grazing or near-tangent rays — precisely the situation this course's own numerical-methods material exists to catch before it becomes an invisible, hard-to-diagnose rendering bug.

Ray-Plane Intersection

For a plane through point `Q` with normal `N`, and a ray `O+tD`: `t = (Q−O)·N / (D·N)`. A zero (or near-zero) denominator means the ray is parallel to the plane.

VERIFIED DIRECTLY

A ray from the origin straight up, `D=(0,0,1)`, against the horizontal plane `z=5` (`Q=(0,0,5)`, `N=(0,0,1)`): `t=5.0`, hitting exactly `(0,0,5)`. The same plane tested against a horizontal ray, `D=(1,0,0)`: `D·N=0` exactly, correctly signaling the ray runs parallel to the plane and never reaches it — the same denominator-as-degeneracy-signal pattern as the line-line case above.

Point-in-Polygon: Extending Chapter 4's Side Test to a Whole Shape

For a **convex** polygon, a point is inside if it's on the same side of *every* edge — exactly Chapter 4's cross-product sign test, applied once per edge and checked for consistency.

VERIFIED DIRECTLY — INSIDE, OUTSIDE, AND THE BOUNDARY EDGE CASE

Testing against the square `(0,0),(4,0),(4,4),(0,4)`: the point `(2,2)` gives all four cross-product signs positive (`8,8,8,8`) — clearly inside. The point `(5,2)` gives mixed signs (`8,−4,8,20`) — clearly outside. The point `(4,2)`, sitting exactly on the right edge, gives one sign of exactly `0` and the rest positive — a genuine boundary case, classified as "inside" only because the test used `≥0` rather than strict `>0`.

A REAL DESIGN DECISION, NOT AN EDGE CASE TO IGNORE

Whether a point exactly on a polygon's boundary counts as "inside" is a genuine choice a real implementation has to make explicitly — and, echoing Numerical Methods & Floating-Point Computation's own repeated theme, a point that's supposed to be exactly on an edge (like a mouse click precisely on a button's border) may not land on exactly `0` once real floating-point coordinates are involved, making a small tolerance around the boundary — not strict `>0`/`<0` — the more robust real-world choice.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT DRAWS ON
Ray-sphere intersection reproducing Numerical Methods & Floating-Point Computation Chapter 4's exact numbers	A direct, concrete real-world application of a prerequisite course's own core finding, not just a passing mention
A zero denominator signaling "parallel," in both line-line and ray-plane tests	The same degeneracy-detection instinct from Numerical Methods & Floating-Point Computation's own conditioning material, applied geometrically
Point-in-polygon's boundary case needing an explicit tolerance decision	Directly echoes Chapter 1's own `0.1+0.2` exact-equality warning, now applied to a geometric test instead of arithmetic

Hands-On Exercises

EXERCISE 1

Using this chapter's own line-line formula, find the intersection point of the line through `(1,1)` and `(5,5)` with the line through `(1,5)` and `(5,1)`.

View solution

EXERCISE 2

Using this chapter's own verified ray-sphere finding, explain specifically why a ray tracer using the naive quadratic formula would tend to produce its worst visible errors on rays that hit a sphere at a shallow, grazing angle rather than a ray that hits the sphere dead-on through its center.

View solution

EXERCISE 3

Using this chapter's own point-in-polygon boundary-case finding, explain why a UI system that checks whether a mouse click landed exactly on a button's edge using strict cross-product signs (rather than a small tolerance) could cause a real, observable bug for the user, and describe what that bug would look like.

View solution

CHAPTER 9 QUICK REFERENCE

- **Line-line intersection:** a determinant-style formula; a zero denominator means parallel lines — verified `(2,2)` crossing point and a verified parallel case
- **Ray-sphere intersection:** substituting the ray into the sphere equation gives a genuine quadratic — verified to reproduce Numerical Methods & Floating-Point Computation Chapter 4's exact numbers ($\approx 3.38\times10^{-7}$ naive vs. $\approx 1.14\times10^{-16}$ stable) in a real geometric setup
- **Ray-plane intersection:** `t=(Q−O)·N/(D·N)`; a zero denominator means the ray is parallel to the plane — verified with a real hit point and a verified parallel case
- **Point-in-polygon (convex):** Chapter 4's cross-product side test, applied to every edge — verified inside/outside/boundary cases, with an honest note that boundary inclusion is a real design decision needing a tolerance, not strict equality
- **Next chapter:** Capstone — building a small 2D/3D geometry toolkit that exercises every chapter in this course together

CHAPTER 10 OF 10

Capstone — Building a Small 2D/3D Geometry Toolkit

Geometry & Trigonometry

Chapter 10 · Capstone — Building a Small 2D/3D Geometry Toolkit

One continuous project: authoring, locating, lighting, animating, orienting, rendering, and finally letting a player click on a single game object — a treasure chest — following it through every chapter this course built, in the order a real object actually moves through a real engine's pipeline. Every step below directly reuses that chapter's own already-verified numbers, rather than re-deriving them, exactly the way a real engineer reuses a formula they've already trusted and tested.

STEP	TASK	CHAPTER(S) USED
1	Author the chest's rotation and convert units	Ch.2
2	Triangulate the chest's distance on the minimap	Ch.3
3	Light the chest and check which side of the fence it's on	Ch.4
4	Animate the chest's idle spin around its own center	Ch.5
5	Let the player freely orient the chest — and hit gimbal lock	Ch.6
6	Fix it with quaternions	Ch.7
7	Place and view the chest through the scene camera	Ch.8
8	Let the player click the chest, and check a tooltip box	Ch.9

Step 1 — Authoring the Chest's Rotation

CH.2

The chest's designer sets its authored rotation to `45°` in the level editor. The engine's own math library works entirely in radians.

VERIFIED DIRECTLY

`45° = π/4 = 0.7853981633974483` radians — confirmed two independent ways (`math.radians(45)` and `math.pi/4` agree exactly). As the chest idly spins every frame afterward, its accumulated rotation angle gets wrapped back into `[0,2π)` every frame — Chapter 2's own verified 2,000,000-frame experiment showed doing this keeps

`sin()` / `cos()` accurate to `≈4.54×10⁻¹¹` relative error, versus `≈3.15×10⁻⁷` if the angle were ever left to grow unbounded.

Step 2 — Triangulating the Chest's Position on the Minimap

CH.3

Two scouts stand `100m` apart and each report the angle to the chest: `60°` and `70°`.

VERIFIED DIRECTLY — REUSING CHAPTER 3'S OWN TRIANGULATION EXACTLY

Law of Sines gives the chest's distance from each scout: `≈122.67m` and `≈113.05m`. Feeding those two distances back through the Law of Cosines recovers the original `100m` baseline almost exactly (`100.00000000000001`) — the same cross-check discipline confirming the minimap's triangulation is self-consistent before trusting it.

Step 3 — Lighting the Chest, and Checking the Fence Line

CH.4

The chest's lid is a triangle with vertices `(0,0,0)`, `(2,0,0)`, `(0,3,1)`. The scene also has a fence running from `(0,0)` to `(4,0)`, and the game needs to know which side of it the chest sits on.

VERIFIED DIRECTLY — REUSING CHAPTER 4'S OWN NORMAL, LIGHTING, AND SIDE-TEST NUMBERS

The lid's unit normal, `(0, -0.3162, 0.9487)`, verified perpendicular to both edges and unit length. Lit from the front: brightness `≈0.6069`. Lit from directly behind (impossible for a real light, but a useful stress test): brightness `≈-0.6069`, correctly clamped to `0`. The chest, sitting at `(2,3)` relative to the fence, gives a cross-product side value of `+12` — the same sign as a known "inside the yard" reference point, confirming it's on the correct side.

Step 4 — The Chest's Idle Spin

CH.5

The chest spins slowly in place as an idle animation — rotating around its own center, not the world origin.

VERIFIED DIRECTLY — REUSING CHAPTER 5'S OWN PIVOT ROTATION AND COMPOSITION FINDINGS

A marker point on the chest's lid, `(5,5)`, rotated `90°` around the chest's own center `(2,2)`, lands at exactly `(-1,5)` — confirming the arbitrary-pivot formula, not the origin, is what the idle-spin code actually uses. Composing two `30°` and `45°` spin increments matches a single `75°` rotation to within `≈10⁻¹⁶`. But updating the chest's rotation matrix this way every single frame, for a very long play session, is exactly the scenario Chapter 5 verified drifts — after `200,000` frame updates, the matrix's determinant creeps to `1.0000000000188298` instead of exactly `1`, subtly distorting the chest's own shape over time if left uncorrected.

Step 5 — Free Orientation Control, and Gimbal Lock

CH.6

In the player's inventory screen, the chest can be freely rotated with three sliders: yaw, pitch, and roll. A tester reports that at one specific pitch value, the yaw and roll sliders "stop doing different things."

VERIFIED DIRECTLY — REUSING CHAPTER 6'S OWN GIMBAL-LOCK FINDING EXACTLY

At `pitch=90°`, four different `(yaw,roll)` combinations that all share `yaw-roll=20°` — `(30°,10°)`, `(40°,20°)`, `(25°,5°)`, `(100°,80°)` — produce the **identical** orientation matrix. The tester's bug report is genuine, reproducible, and exactly this: at that one pitch value, the yaw and roll sliders really have collapsed into controlling the same thing.

Step 6 — Fixing It With Quaternions

CH.7

The inventory screen's orientation control is switched from Euler-angle sliders to an internal quaternion representation, driven by a virtual trackball instead.

VERIFIED DIRECTLY — REUSING CHAPTER 7'S OWN QUATERNION FINDINGS EXACTLY

Quaternion composition reproduces Chapter 6's own matrix results exactly — the internal representation genuinely changed, the rotations themselves didn't. Composing rotations through the exact orientation that broke the slider UI stays numerically smooth, with no coordinate singularity. The one honest limit still applies: if the UI ever needs to display numeric yaw/pitch/roll values again (say, in a debug overlay), converting the quaternion back still collapses `(30°,10°)` and `(40°,20°)` to the identical displayed numbers — switching representations fixed the control scheme, not the underlying geometric fact about that orientation.

Step 7 — Placing and Viewing the Chest Through the Camera

CH.8

The chest, at world position `(5,2)` rotated `30°`, needs to be drawn from the scene's camera, sitting at world position `(10,10)` rotated `-45°`.

VERIFIED DIRECTLY — REUSING CHAPTER 8'S OWN FULL PIPELINE EXACTLY

The chest's local forward-marker `(1,0)` transforms to world position `(5.866025403784438, 2.5)`, then into camera space at `(2.380139..., -8.226462...)` — matching exactly whether computed as two separate steps or as one precomputed combined matrix (`0.0` difference). The camera's own inverse transform, checked against its forward transform, multiplies back out to the exact identity matrix — confirming the whole pipeline is self-consistent before a single pixel gets drawn.

Step 8 — The Player Clicks the Chest

CH.9

The player clicks on the chest. The engine casts a ray from the camera through the click point and tests it against the chest's bounding sphere — and separately checks whether the same click also landed inside a nearby tooltip box.

VERIFIED DIRECTLY — REUSING CHAPTER 9'S OWN RAY-SPHERE AND POINT-IN-POLYGON FINDINGS EXACTLY

The ray-sphere test, set up exactly as Chapter 9 verified (`a=1, b=-100000, c=1` — a ray originating far from a large bounding volume, a completely ordinary real setup), shows the naive quadratic formula's near-hit-point relative error at `≈3.38×10⁻⁷` versus the stable formula's `≈1.14×10⁻¹⁶` — confirming the pick-testing code must use the stable reformulation, not the textbook one, to reliably report exactly where the click landed on the chest's surface. Separately, the same click point tested against the tooltip's boundary box reuses Chapter 9's own boundary-case finding: a click landing exactly on the tooltip's edge needs a small tolerance, not strict inequality, to behave predictably.

What This Course Doesn't Cover

As stated honestly back in Chapter 1: formal proof-based Euclidean geometry, projective and non-Euclidean geometry, and differential geometry/manifolds were all named as deliberately out of scope, and stayed out of scope through all ten chapters. This capstone built and picked one simple object; a real engine repeats exactly

this pipeline for every object in a scene, every frame — the underlying mathematics doesn't change with scale, only the bookkeeping does.

Where This Course Connects

Linear Algebra Fundamentals' own vectors, dot/cross products, and its brief rotation-matrix introduction underwrote nearly every chapter here directly. Numerical Methods & Floating-Point Computation's own catastrophic-cancellation and stable-quadratic-formula material was reused explicitly and by name in Chapters 4 and 9 — this course's own capstone (Step 8) is a direct, concrete application of that course's own capstone lesson, applied to a genuinely different problem. Calculus & Optimization's derivative and chain-rule material underlies the smooth interpolation (SLERP) briefly mentioned in Chapter 7, a natural next step beyond this course's own scope.

Hands-On Exercises

EXERCISE 1

Using this chapter's own Step 3 lighting numbers, explain what would happen to the chest's rendered lid if the game's lighting code forgot the `max(0, ...)` clamp verified back in Chapter 4, in a scene with two lights, one in front of the chest and one behind it.

View solution

EXERCISE 2

Using this chapter's own Step 5 and Step 6, explain why switching the inventory screen's orientation control from Euler-angle sliders to a quaternion-driven trackball fixes the tester's bug report, but would **not** fix a hypothetical second bug report about a debug overlay that displays the chest's numeric yaw/pitch/roll values.

View solution

EXERCISE 3

Using this chapter's own Step 8, explain in your own words why the engine needs *both* the numerically stable ray-sphere formula *and* a boundary tolerance for the tooltip check — that is, why fixing only one of the two would still leave a real, user-visible bug in the picking system.

View solution

CHAPTER 10 QUICK REFERENCE

- **Full worked project:** author + convert units (Ch.2) → triangulate position (Ch.3) → light + side-test (Ch.4) → idle-spin around own pivot (Ch.5) → free orientation + gimbal lock (Ch.6) → fix with quaternions (Ch.7) → full render pipeline (Ch.8) → mouse-pick with a stable ray-sphere test + tolerant boundary check (Ch.9)
- Every step directly reused that chapter's own already-verified numbers, rather than re-deriving them from scratch — exactly how a real engineer builds on trusted, tested formulas
- The one recurring theme across all eight steps: geometry code is only as reliable as its numerical foundations — a correct formula used carelessly (naive quadratic roots, exact-equality boundary checks, unwrapped angles, un-renormalized rotations) still produces real, user-visible bugs
- Out of scope: formal proof-based Euclidean geometry, projective/non-Euclidean geometry, differential geometry/manifolds
- **Course complete** — Geometry & Trigonometry, 10 chapters, from a single naive heading-subtraction bug to a fully picked, rendered, and orientable game object