

Graphics Programming

From the dot product to **hardware ray tracing**:
C++23 and **Metal 4**, with every picture rendered
by the code in the book.

```
shadow.metal - Metal Shading Language

kernel void shadowRays(instance_acceleration_structure scene,
                        uint2 gid [[thread_position_in_grid]]) {
    ray r; r.origin = P + N * 1e-3; r.direction = toLight;
    intersector<instancing> isect;
    isect.accept_any_intersection(true);
    auto hit = isect.intersect(r, scene);
    shadow.write(hit.type == intersection_type::none ? 1.0 : 0.0, gid);
}
```

Contents

0	Before You Start	5
0.1	How to read this guide	5
0.2	What each badge promises	5
0.3	Conventions used everywhere	6
0.4	Your toolchain	6
1	How a Frame Gets Made	8
1.1	From a scene to a grid of colors	8
1.2	The real-time pipeline	8
1.3	CPU, GPU and the frame	9
1.4	Apple GPUs are tile-based	10
1.5	What one frame of a modern engine contains	10
2	Vectors	12
2.1	Points, directions and components	12
2.2	Length, scaling and normalizing	12
2.3	The dot product	13
2.4	The cross product	13
2.5	Reflection	14
2.6	Vectors in C++23	14
2.7	Orthonormal bases	15
3	Matrices and Transforms	16
3.1	A matrix is a transformation	16
3.2	Homogeneous coordinates: one matrix for everything	16
3.3	The transforms you need	17
3.4	Order matters	17
3.5	Matrices in C++23	17
3.6	Rotations: Euler angles, axis-angle and quaternions	19
3.7	Transforming normals: the inverse-transpose	19
4	Cameras and Projection	21
4.1	The journey of a vertex	21
4.2	The view matrix	21
4.3	Perspective: similar triangles	22
4.4	Depth precision, and why you want reversed-Z	23
4.5	A vertex's whole trip, with numbers	24
4.6	A camera you can fly	25
4.7	Orthographic projection and the viewport	26
5	Rasterization	27
5.1	Which pixels does a triangle cover?	27
5.2	Barycentric coordinates	27

5.3	Perspective-correct interpolation	29
5.4	Clipping	30
5.5	The depth buffer	31
5.6	Back-face culling and winding	31
5.7	Anti-aliasing: coverage at sub-pixel level	32
5.8	Quads, derivatives and the cost of small triangles	33
6	Textures and Sampling	34
6.1	UV coordinates and texel centers	34
6.2	Filtering when the texture is magnified	34
6.3	Minification and aliasing: mipmaps	35
6.4	Trilinear and anisotropic filtering	36
6.5	Color spaces and pixel formats	36
6.6	Creating and binding a texture in Metal 4	37
6.7	Sampling in a shader	38
7	Light and Color	40
7.1	Measuring light	40
7.2	Why there is a cosine: Lambert's law	40
7.3	Light falls off with distance	41
7.4	Gamma, sRGB and linear light	41
7.5	HDR, exposure and tone mapping	43
8	Materials and PBR	44
8.1	The BRDF: how a surface reflects	44
8.2	Classic models	44
8.3	Microfacet theory	45
8.4	The metalness workflow	46
8.5	The complete shading function	47
8.6	Is it energy conserving? The furnace test	47
9	Metal 4 Fundamentals	50
9.1	What changed in Metal 4	50
9.2	The object model at a glance	51
9.3	Residency: the GPU may only touch memory you declared	52
9.4	The frame loop and CPU-GPU synchronization	52
9.5	Barriers: saying which work waits for which	53
9.6	Ask the device what it supports	54
10	Your First Pipeline	55
10.1	What a render pipeline is	55
10.2	Vertex pulling: no vertex descriptor	55
10.3	Structs must match on both sides	55
10.4	Shaders	56
10.5	Building the pipeline and depth state	56
10.6	Per-draw data: the upload ring	57

11	Compute and Parallelism	59
11.1	The GPU execution model	59
11.2	Pick a threadgroup size	59
11.3	Writing the compute pipeline in Metal 4	60
11.4	A worked algorithm: parallel prefix sum	61
12	Monte Carlo Rendering	63
12.1	Integrals you cannot solve by hand	63
12.2	Importance sampling: aim your samples	63
12.3	Random numbers and sampling helpers on the GPU	65
12.4	From integral to path tracer	65
13	Ray Tracing	68
13.1	A ray and a triangle	68
13.2	Why you need an acceleration structure	69
13.3	Hardware ray tracing on Apple GPUs	71
13.4	Tracing from a compute shader: soft shadows	73
13.5	Ambient occlusion and global illumination	74
14	The Render Graph	76
14.1	The problem	76
14.2	What the graph does	76
14.3	A working render graph	77
14.4	The G-buffer and debug views	79
15	Debugging and Performance	81
15.1	A debugging method for graphics	81
15.2	Tools	81
15.3	Performance on a tile-based GPU	82
15.4	Where ARM64 helps: the CPU side of a Metal engine	83
15.5	Upgrade checklist for a physically based renderer	85
16	Reference Pages	86
16.1	Conventions used throughout	86
16.2	Frame recipe (Metal 4)	86
16.3	MSL quick reference	87
16.4	Formulas on one page	87
16.5	Glossary	88

Before You Start

How this guide works, what the badges on each listing mean, the conventions every chapter shares, and the tools you need.

0.1 How to read this guide

Graphics programming is three subjects braided together: **math** (vectors, matrices, probability), **rendering ideas** (how a pile of triangles becomes light on a screen) and **API code** (telling a GPU what to do). This guide teaches them together, in an order where each chapter needs only the ones before it.

- Chapters 1–4** The pipeline, then the math: vectors, matrices, cameras. Read these in order.
- Chapters 5–8** Rasterization, textures, light, and physically based materials. Every figure here was rendered by the C++ shown beside it.
- Chapters 9–11** Metal 4: the object model, your first pipeline, then compute and the GPU's parallel model.
- Chapters 12–14** Monte Carlo, path tracing, and hardware ray tracing: shadows, ambient occlusion and global illumination.
- Chapters 15–16** Render graphs, frame architecture, debug views, and how to find out why your frame is slow or wrong.

Remember • A tip box holds a rule of thumb you will use constantly.

Pitfall • A pitfall box marks a classic trap, usually one that produces a black screen or a subtle artifact.

Math • This color marks the key math result of a section: the thing worth memorizing.

Reading a code box

✓ compiled & run

```
// comments are gray italics
float3 n = normalize(v);
[[buffer(0)]] // attributes
```

OUTPUT

this dark strip is real program output

0.2 What each badge promises

I had no Mac or GPU while writing this, so I did not want a single claim to hide behind a polished page. Each listing is stamped with exactly how far it was checked:

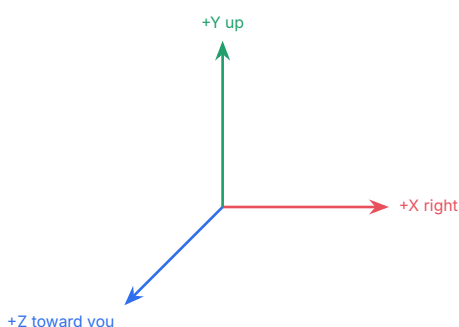
- ✓ **compiled & run** Built with `g++ 13 -std=c++23 -Wall -Wextra -pedantic` with zero warnings, executed, and the real output pasted below it. Every picture in chapters 5–8 and 12–13 comes from programs like these.
- ✓ **type-checked** Metal host code in C++. Compiled (syntax and types only) by clang 18 against the *real* metal-cpp headers from Apple's repository; Apple's system headers were stubbed. This catches wrong names, wrong argument types and missing includes. It does **not** prove the code behaves correctly on a GPU: nothing here was ever run on one.
- ✓ **MSL, CPU-run** Metal Shading Language functions that use only math. I ran the identical text on the CPU through a small compatibility header that defines `float3`, `saturate` and friends, and tested the results against reference values.
- ◇ **not compiled here** Shaders that need Apple's compiler (ray-tracing intrinsics, textures, stage attributes). Written carefully from the documented API; treat them as a strong draft and expect to fix a typo or two.

Version caveat • Metal 4 arrived with macOS 26. The metal-cpp snapshot I checked against (mid-2026) already contains newer macOS 27 API surface, and newer *shading-language* features (cooperative tensors, low-precision matrix multiply) are reported to need macOS 27. I mention them only as pointers, and not in the main code. Always gate features with the capability queries of chapter 9.

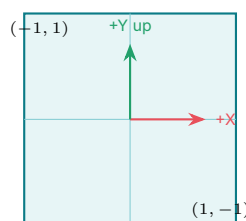
0.3 Conventions used everywhere

Graphics bugs are very often a disagreement about conventions: handedness, row versus column, which way is up. Fix them once, here, and every formula in the guide obeys them. They match Metal's own.

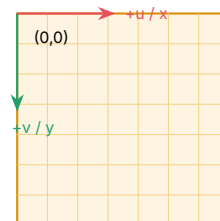
THE FOUR COORDINATE SPACES YOU WILL KEEP MEETING



World / view space
right-handed; the camera looks down $-Z$



NDC $x, y \in [-1, 1]$
 $z \in [0, 1]$ (0 = near)



Texture & framebuffer
origin top-left, y grows down

Topic	Convention in this guide (and in Metal)
Handedness	Right-handed world and view space; $\vec{x} \times \vec{y} = +\vec{z}$.
Vectors & matrices	Column vectors; transforms multiply on the <i>left</i> : $\vec{p}' = M\vec{p}$.
Storage	Column-major: element (row r , column c) is at index $4c + r$. In C++23 we write it <code>M[r, c]</code> .
Clip-space depth	$z \in [0, 1]$ after the divide (OpenGL uses $[-1, 1]$: do not copy its matrices).
Winding	Counter-clockwise (seen from outside) is the front face.
Colors	Math happens in <i>linear</i> light; storage and display use sRGB (chapter 7).
Units	Meters, radians, floats. Angles in code are radians; degrees appear only in prose.

0.4 Your toolchain

You need a Mac with Apple silicon and a recent Xcode (for the Metal compiler and the GPU debugger). Hardware ray tracing needs an M3-class GPU or newer; the capability query below tells you what you actually have. metal-cpp is a header-only C++ wrapper over Metal: download it from Apple's developer site or GitHub and put it on your include path.

Ask the device what it supports (needs the includes from chapter 9)

✓ type-checked against Apple's real metal-cpp headers - never run on a GPU

```
#include <stdio>

// Ask the device what it can do before choosing a code path. Never assume from the marketing name.
void printCaps(MTL::Device* d) {
    std::printf("%s\n", d->name()->utf8String());
    std::printf("  Metal 4 family      : %d\n", d->supportsFamily(MTL::GPUFamilyMetal4));
    std::printf("  Apple9+ (M3 class)  : %d\n", d->supportsFamily(MTL::GPUFamilyApple9));
    std::printf("  Apple10             : %d\n", d->supportsFamily(MTL::GPUFamilyApple10));
    std::printf("  ray tracing         : %d (from render stages: %d)\n",
        d->supportsRaytracing(), d->supportsRaytracingFromRender());
    std::printf("  unified memory      : %d\n", d->hasUnifiedMemory());
    std::printf("  working set         : %.1f GiB\n", d->recommendedMaxWorkingSetSize() / 1073741824.0);
}
```

```
}
```

Build commands (macOS terminal)

```
# 1. compile the shaders into a library the app loads at runtime
xcrun -sdk macosx metal -c shaders.metal -o shaders.air
xcrun -sdk macosx metallib shaders.air -o shaders.metallib

# 2. compile and link the app (metal-cpp is headers only)
clang++ -std=c++2b -O2 -I metal-cpp main.cpp \
    -framework Metal -framework Foundation -framework QuartzCore \
    -framework AppKit -o app
```

Where does the window come from? • metal-cpp wraps Metal, not AppKit. You need a few lines of Objective-C++ (or SDL/GLFW) to make a window and give you a `CAMetalLayer`. After that, everything in this guide is plain C++23. This guide keeps the window code out of the way and starts from the layer.

How a Frame Gets Made

The big picture before any math: rasterization versus ray tracing, the real-time pipeline, who does what on the CPU and GPU, and how Apple's GPUs differ.

1.1 From a scene to a grid of colors

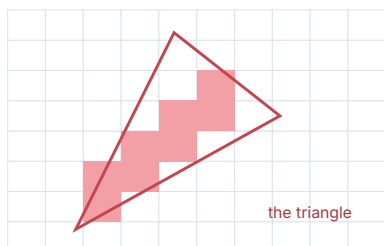
A renderer takes a **scene**—triangles with materials, some lights, a camera—and produces an **image**: a grid of pixels, each holding a color. Everything in this guide is a way of answering one question for each pixel: *what light travels from the scene to the camera through this pixel?*

There are two families of answers, and the difference is just which loop is on the outside.

TWO WAYS TO FILL THE SAME GRID: WHICH LOOP IS OUTERMOST?

Rasterization

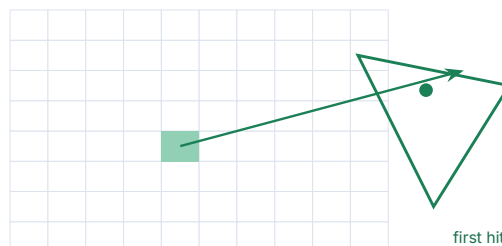
```
for each triangle:
  for each pixel it covers:
    shade, keep if closest
```



Work is proportional to the triangles' *screen area*. Fast, predictable, and the foundation of all real-time graphics.

Ray tracing

```
for each pixel:
  shoot a ray into the scene
  find the closest triangle hit
```



Work is proportional to the *pixels* (times rays per pixel). Reflections, shadows and global light fall out naturally, but need an acceleration structure.

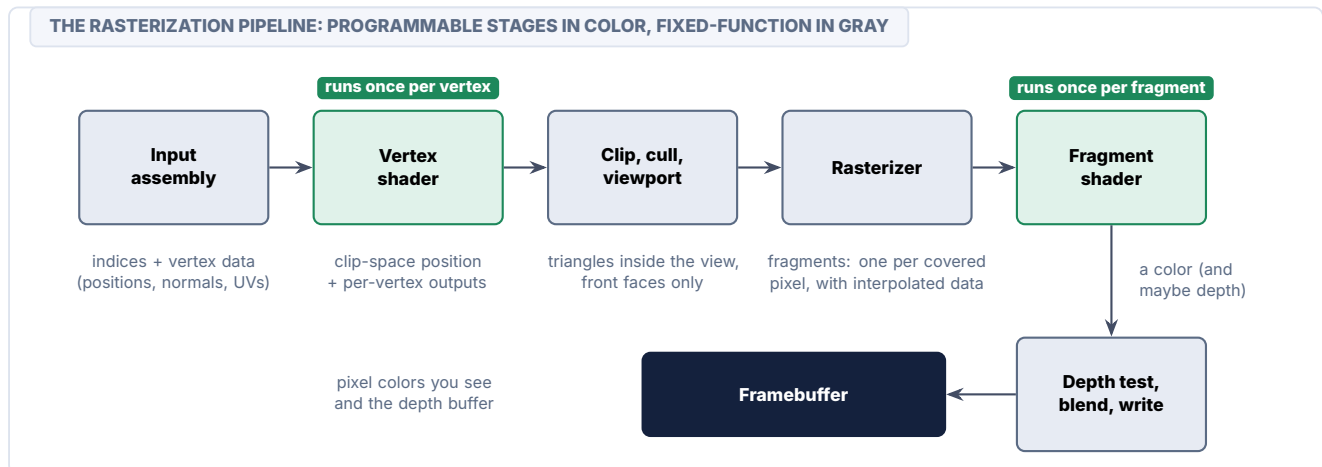
Real engines use both. Rasterization draws the primary view cheaply; ray tracing then answers the questions rasterization finds awkward: is this point in shadow? what is reflected here? how much light bounces in from the room? Metal 4 on M3-class and newer GPUs has hardware that accelerates exactly these ray queries (chapters 13 and 14).

1.2 The real-time pipeline

Rasterizing a triangle is a conveyor belt. Some stages you program (colored below) and some are fixed-function hardware you only configure. The data format changes at every stage, and it is worth knowing the names.

- **Vertex shader** (once per vertex): turn a vertex's model-space position into a clip-space position (chapters 3–4). It can also pass any per-vertex values along.
- **Rasterizer**: decides which pixels each triangle covers and *interpolates* the vertex outputs across it (chapter 5). Fixed-function hardware.
- **Fragment shader** (once per covered pixel): compute the color. This is where lighting and materials (chapters 7–8) live. Called a *pixel shader* in Direct3D.

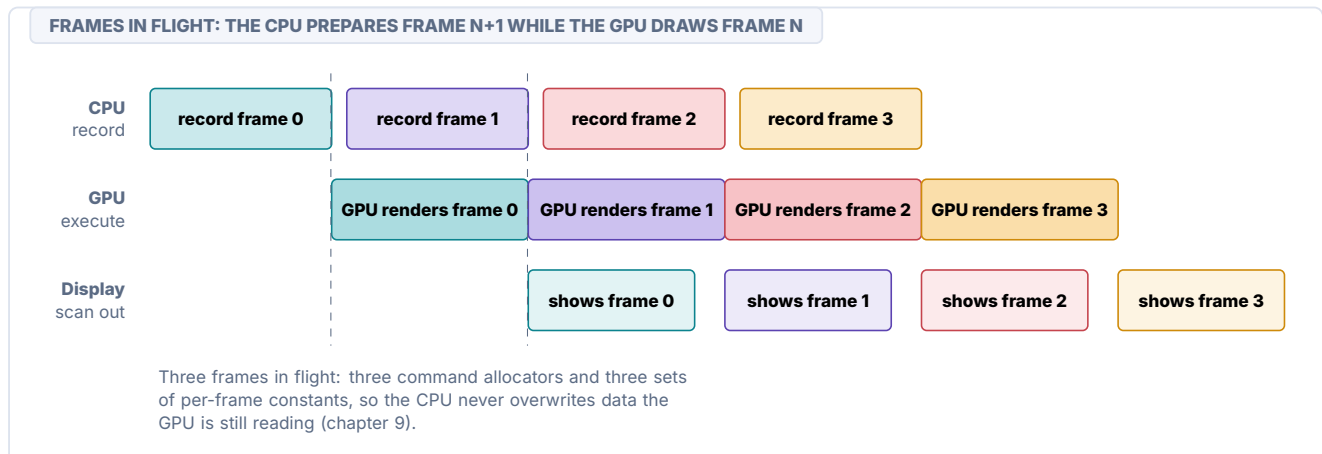
- **Depth test and blend:** keep the nearest surface; optionally mix with what is already there.



Modern variant • Metal also has *mesh shaders*: an object stage and a mesh stage replace input assembly and the vertex shader, letting the GPU generate and cull geometry itself (they replace tessellation). Everything downstream of them in the figure is unchanged.

1.3 CPU, GPU and the frame

The CPU never draws anything. It *records commands* (“set this pipeline, bind these buffers, draw 36 indices”) into a command buffer and submits it; the GPU executes them later, asynchronously. This split drives most of the API’s design.



Frame budget

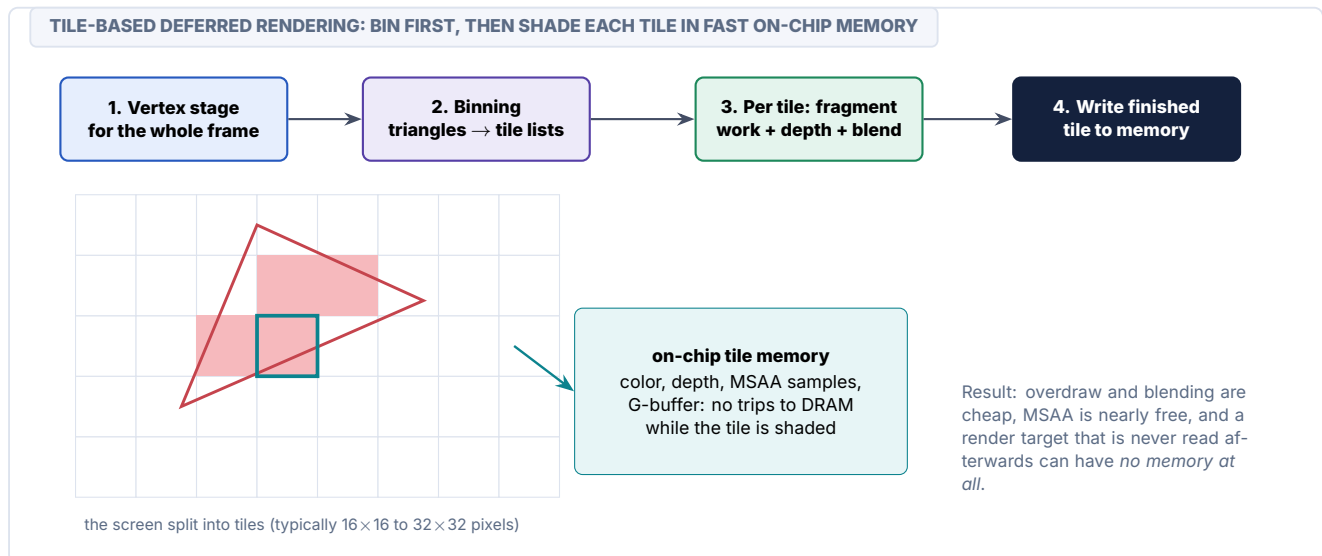
Everything must fit inside one display interval. These numbers are the unit of performance work:

Target	60 fps	90 fps	120 fps	240 fps
Time per frame	16.67 ms	11.11 ms	8.33 ms	4.17 ms

Because $\text{ms} = 1000/\text{fps}$, going from 60 to 120 fps does not mean “twice as fast”; it means *removing* 8.3 ms from every frame. Chapter 16 shows how to find where those milliseconds go.

1.4 Apple GPUs are tile-based

Desktop GPUs from other vendors are *immediate-mode*: they shade triangles in submission order and write each result to video memory. Apple's GPUs are *tile-based deferred renderers* (TBDR), and the memory is *unified* (CPU and GPU share one pool; no upload step). Knowing this changes how you design passes.

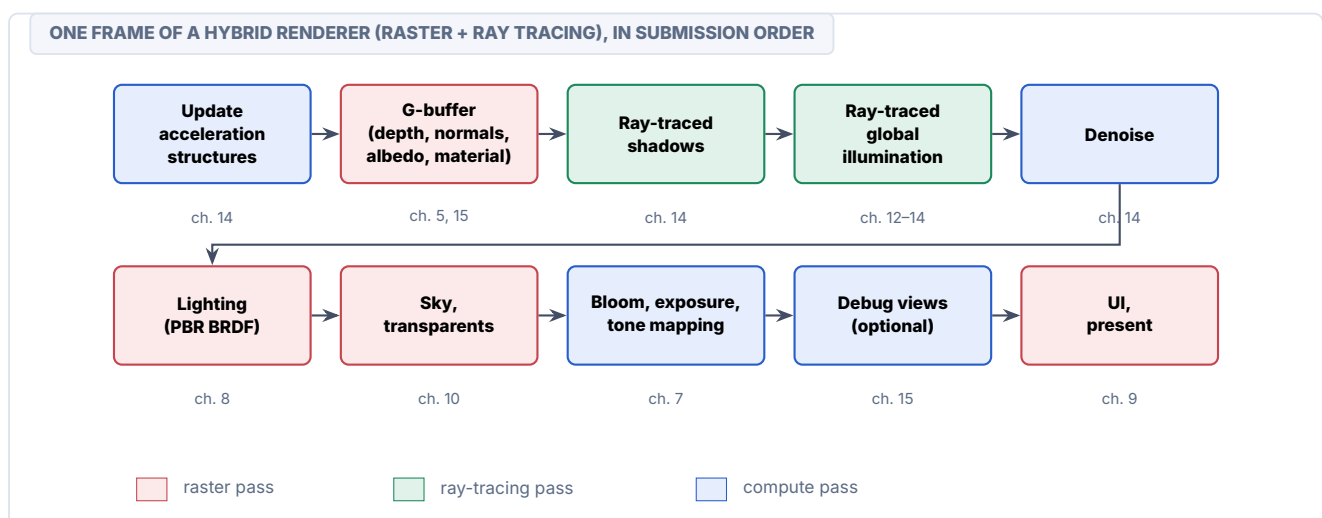


Practical consequences you will meet again in chapters 9 and 15:

- Each render-pass attachment has a **load action** and a **store action**. `Clear` costs nothing; `Load` reads the old contents from memory into the tile. `DontCare / Store` decide whether the finished tile is written back at all. Depth you never sample later should use `DontCare`.
- Hardware hidden-surface removal means opaque overdraw is often not shaded at all; sorting opaque draws front-to-back still helps less than on other GPUs.
- Splitting one pass into several (for example, G-buffer then lighting as separate encoders) forces tiles out to DRAM and back. Apple's tile shaders and programmable blending exist to avoid exactly that.

1.5 What one frame of a modern engine contains

Here is the frame you will have built by the end of this guide. Each box names the chapter that teaches it.



Chapter 1 in one minute

- A renderer answers “what color is this pixel?” either per triangle (rasterization) or per pixel (ray tracing); real engines combine both.
- The rasterization pipeline is vertex shader → clip/cull → rasterize → fragment shader → depth/blend; you program two of those stages.
- The CPU records commands; the GPU executes them later, so you keep several frames in flight.
- Apple GPUs are tile-based with unified memory: choose load/store actions deliberately and avoid needless pass splits.

Vectors

The first of three math chapters. Points, directions, the dot and cross products and reflection: the handful of operations that almost every shader line is made of.

02

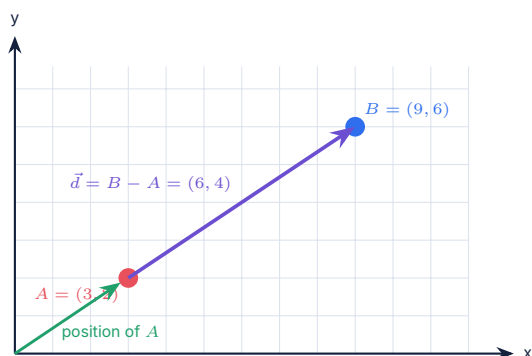
2.1 Points, directions and components

A **vector** is a list of numbers, usually three: $\vec{v} = (x, y, z)$. It can mean two different things, and graphics code constantly mixes them up:

- a **point**: a *position* in space (where the corner of a triangle is);
- a **direction** (or displacement): an *arrow* with a length and a heading (which way the surface faces, which way the light comes from).

They look identical in memory but behave differently. You can subtract two points to get the direction between them, but adding two points is meaningless. Chapter 3 will give them different fourth components ($w = 1$ for points, $w = 0$ for directions) so that matrices treat them correctly.

POINTS VERSUS DIRECTIONS



Points say where: A and B .

Directions say which way and how far: $\vec{d}$. The same $\vec{d}$ drawn anywhere is the same vector.

point - point = direction
point + direction = point
direction + direction = direction

2.2 Length, scaling and normalizing

The **length** (magnitude) of $\vec{v}$ comes from Pythagoras. A vector of length 1 is a **unit vector**; lighting math assumes directions are unit vectors, and **normalizing** (dividing by the length) is how you make them so.

$$|\vec{v}| = \sqrt{x^2 + y^2 + z^2}, \quad \hat{v} = \frac{\vec{v}}{|\vec{v}|}, \quad k\vec{v} = (kx, ky, kz), \quad \vec{a} \pm \vec{b} = (a_x \pm b_x, \dots)$$

Pitfall • Interpolating unit vectors (normals across a triangle) does not give a unit vector, so *re-normalize in the fragment shader*. And never normalize a zero vector: $0/0$ is NaN, and one NaN pixel can spread through a blur or a denoiser into a black square.

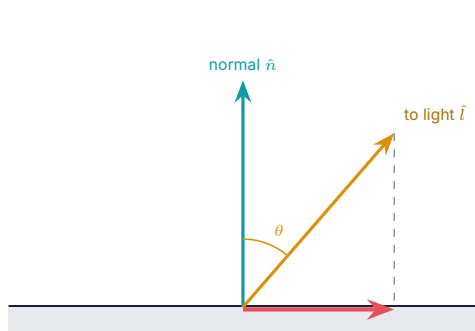
2.3 The dot product

The dot product turns two vectors into one number that measures how much they *agree*:

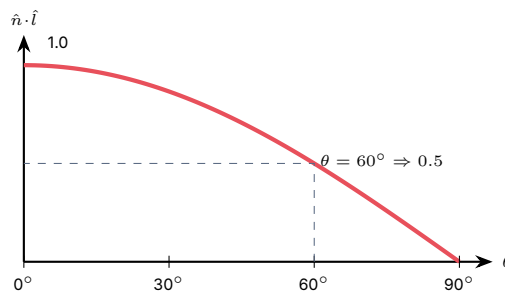
$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z = |\vec{a}| |\vec{b}| \cos \theta$$

For unit vectors it is simply $\cos \theta$: +1 when they point the same way, 0 when perpendicular, -1 when opposite. That one fact powers most of lighting.

THE DOT PRODUCT AS "HOW MUCH LIGHT ARRIVES": LAMBERT'S COSINE LAW



Light arriving at angle θ spreads over a wider patch of surface, so each bit of surface gets less: brightness $\propto \cos \theta = \hat{n} \cdot \hat{l}$.



Where you will use the dot product

- **Diffuse lighting:** $\max(0, \hat{n} \cdot \hat{l})$. The max clamps light arriving from behind the surface.
- **Facing tests:** $\hat{n} \cdot \hat{v} < 0$ means a surface faces away from the viewer (back-face culling).
- **Projection:** $\vec{a} \cdot \hat{b}$ is the length of $\vec{a}$'s shadow along $\hat{b}$; this is how the view matrix measures coordinates (chapter 4).
- **Angles:** $\theta = \arccos(\hat{a} \cdot \hat{b})$. Avoid it in shaders; compare dot products directly.

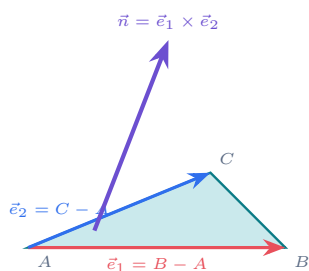
2.4 The cross product

The cross product takes two 3D vectors and returns a third that is **perpendicular to both**, with the right-hand rule choosing the sign (curl your fingers from $\vec{a}$ toward $\vec{b}$; your thumb points along $\vec{a} \times \vec{b}$).

$$\vec{a} \times \vec{b} = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x), \quad |\vec{a} \times \vec{b}| = |\vec{a}| |\vec{b}| \sin \theta$$

Its length is the area of the parallelogram the two vectors span. That gives the two jobs you will use it for: **surface normals** and **areas**.

CROSS PRODUCT: THE NORMAL AND THE AREA OF A TRIANGLE (RIGHT-HAND RULE)



Triangle normal: $\hat{n} = \text{normalize}((B - A) \times (C - A))$ for vertices listed counter-clockwise as seen from outside the surface.

Triangle area: $\frac{1}{2} |(B - A) \times (C - A)|$. The rasterizer uses the same quantity (chapter 5) to compute barycentric weights.

Swap two vertices and the normal flips: that is exactly why winding order decides which side is the front face.

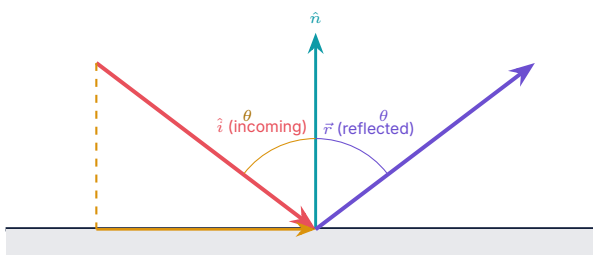
2.5 Reflection

When a ray with direction $\hat{i}$ (pointing *toward* the surface) hits a surface with unit normal $\hat{n}$, the mirror direction is

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

The dot product measures how far $\hat{i}$ pokes into the surface; subtracting twice that component flips it. The same formula gives mirror reflections, specular highlights and the bounce direction of a ray tracer.

REFLECTING A DIRECTION ABOUT THE NORMAL



The angle in equals the angle out. In code, `reflect(i, n)` does this for you.

Refraction (light bending into glass or water) uses Snell's law, $n_1 \sin \theta_1 = n_2 \sin \theta_2$; the Fresnel equations (chapter 8) say how much reflects and how much refracts.

2.6 Vectors in C++23

Here is the whole vector library the rest of the guide builds on, 28 lines. It is plain C++: no SIMD tricks, no templates. `constexpr` lets the compiler evaluate it at compile time when the inputs are constants.

gfx_vec.h: a minimal 3D vector type

✓ header · compiled & used by the demo that follows

```
// gfx_vec.h -- 3D vectors (right-handed, float)
#pragma once
#include <cmath>
#include <numbers>

struct Vec3 {
    float x = 0, y = 0, z = 0;
    constexpr Vec3 operator-() const { return {-x, -y, -z}; }
    constexpr Vec3& operator+=(Vec3 o) { x += o.x; y += o.y; z += o.z; return *this; }
};

constexpr Vec3 operator+(Vec3 a, Vec3 b) { return {a.x + b.x, a.y + b.y, a.z + b.z}; }
constexpr Vec3 operator-(Vec3 a, Vec3 b) { return {a.x - b.x, a.y - b.y, a.z - b.z}; }
constexpr Vec3 operator*(Vec3 a, float s) { return {a.x * s, a.y * s, a.z * s}; }
constexpr Vec3 operator*(float s, Vec3 a) { return a * s; }
constexpr Vec3 operator/(Vec3 a, float s) { return a * (1.0f / s); }

constexpr float dot(Vec3 a, Vec3 b) { return a.x * b.x + a.y * b.y + a.z * b.z; }
constexpr Vec3 cross(Vec3 a, Vec3 b) {
    return {a.y * b.z - a.z * b.y,
            a.z * b.x - a.x * b.z,
            a.x * b.y - a.y * b.x};
}

inline float length(Vec3 a) { return std::sqrt(dot(a, a)); }
inline Vec3 normalize(Vec3 a) { return a / length(a); }
constexpr Vec3 reflect(Vec3 i, Vec3 n) { return i - 2.0f * dot(i, n) * n; }
```

Using it: perpendicular vectors, angles, reflection, triangle area

✓ compiled & run · g++13, -std=c++23 -Wall -Wextra -pedantic

```
#include <cstdio>
#include "gfx_vec.h"

int main() {
    Vec3 x{1, 0, 0}, y{0, 1, 0};
    Vec3 z = cross(x, y); // right-handed: x cross y = +z
    std::printf("x cross y = (%g, %g, %g)\n", z.x, z.y, z.z);
    std::printf("x dot y = %g (perpendicular)\n", dot(x, y));
}
```

```
Vec3 a = normalize({1, 1, 0});
float angle = std::acos(dot(x, a)) * 180.0f / std::numbers::pi_v<float>;
std::printf("angle(x, a) = %.1f degrees\n", angle);

Vec3 n{0, 1, 0}; // floor normal
Vec3 in = normalize({1, -1, 0}); // ray heading down-right
Vec3 out = reflect(in, n);
std::printf("reflect = (%.3f, %.3f, %.3f)\n", out.x, out.y, out.z);

Vec3 e1{2, 0, 0}, e2{0, 3, 0}; // triangle edges
std::printf("tri area = %g\n", 0.5f * length(cross(e1, e2)));
}
```

OUTPUT

```
x cross y = (0, 0, 1)
x dot y = 0 (perpendicular)
angle(x, a) = 45.0 degrees
reflect = (0.707, 0.707, 0.000)
tri area = 3
```

Pitfall: float3 is 16 bytes in MSL • On the GPU, `float3` occupies 16 bytes (12 of data, 4 of padding) and is 16-byte aligned, but the C++ `Vec3` above is 12 bytes. Put one inside a struct shared between C++ and a shader, and every later field lands in the wrong place. Use `packed_float3` (12 bytes) in the shader or add explicit padding in C++. Chapter 10 draws the layouts.

2.7 Orthonormal bases

Often you have just a normal $\hat{n}$ and need two more perpendicular axes: to sample a hemisphere around it, to build a tangent frame for normal mapping, or to place a disc light. A set of three mutually perpendicular unit vectors is an **orthonormal basis**; it defines a local coordinate system. Chapter 12 gives a branchless routine (`buildBasis`) that makes $\hat{t}, \hat{b}$ from $\hat{n}$ without ever picking an arbitrary up".

Cheat sheet: vector identities

$$\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a}, \quad \vec{a} \cdot (\vec{b} + \vec{c}) = \vec{a} \cdot \vec{b} + \vec{a} \cdot \vec{c}$$

dot is symmetric and distributes

$$\vec{a} \times \vec{b} = -\vec{b} \times \vec{a}, \quad \vec{a} \times \vec{a} = \vec{0}$$

cross is anti-symmetric; parallel vectors give zero

$$|\vec{a} \times \vec{b}|^2 + (\vec{a} \cdot \vec{b})^2 = |\vec{a}|^2 |\vec{b}|^2$$

handy sanity check

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

signed volume of the parallelepiped (the determinant of the 3×3 matrix)

$$\hat{n} \cdot (P - P_0) = 0$$

the plane through P_0 with normal $\hat{n}$

$$\vec{a}_{\parallel} = (\vec{a} \cdot \hat{b})\hat{b}, \quad \vec{a}_{\perp} = \vec{a} - \vec{a}_{\parallel}$$

split a vector along and across $\hat{b}$

Chapter 2 in one minute

- Points say where; directions say which way. Subtract points to get directions.
- $\hat{a} \cdot \hat{b} = \cos \theta$: the workhorse of lighting and facing tests. Clamp negatives with $\max(0, \cdot)$.
- $\vec{a} \times \vec{b}$ is perpendicular to both, with length equal to the parallelogram's area: normals and triangle areas.
- $\vec{r} = \hat{i} - 2(\hat{i} \cdot \hat{n})\hat{n}$ reflects. Normalize before you rely on unit length.

Matrices and Transforms

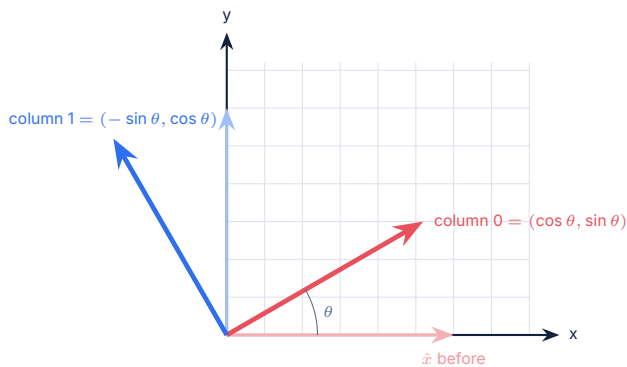
How a single 4×4 grid of numbers can move, turn and scale anything: homogeneous coordinates, composing transforms in the right order, and the normal-matrix trap.

03

3.1 A matrix is a transformation

A matrix is a grid of numbers that **transforms** a vector: $\vec{p}' = M\vec{p}$. The key way to read one: *each column is where an axis ends up*. Feed in $(1, 0)$ and you get column 0; feed in $(0, 1)$ and you get column 1. Everything else is a mix of those.

READING A MATRIX BY ITS COLUMNS: A 30° ROTATION IN 2D



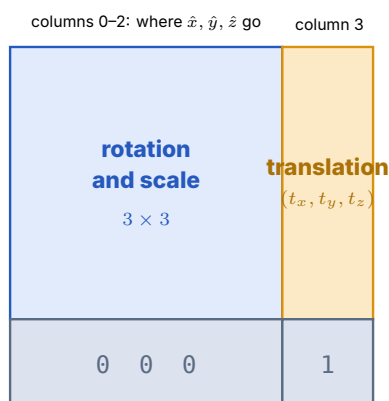
$$R = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}$$

The first column is where $\hat{x}$ lands, the second is where $\hat{y}$ lands. In 3D the same idea gives three columns. This is why you can read a camera's axes straight out of its matrix.

3.2 Homogeneous coordinates: one matrix for everything

A 3×3 matrix can rotate and scale but not *translate* (add an offset), because translation is not linear. The trick is a fourth coordinate w : write a point as $(x, y, z, 1)$ and a direction as $(x, y, z, 0)$. Then a 4×4 matrix can do rotation, scale *and* translation in a single multiply, and directions are (correctly!) unaffected by translation because their $w = 0$ kills the offset.

ANATOMY OF A 4×4 TRANSFORM



$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = M \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Point, $w = 1$: translated too.

Direction, $w = 0$: translation ignored.

A projection matrix (chapter 4) puts something other than $(0, 0, 0, 1)$ in the bottom row; that is what creates perspective.

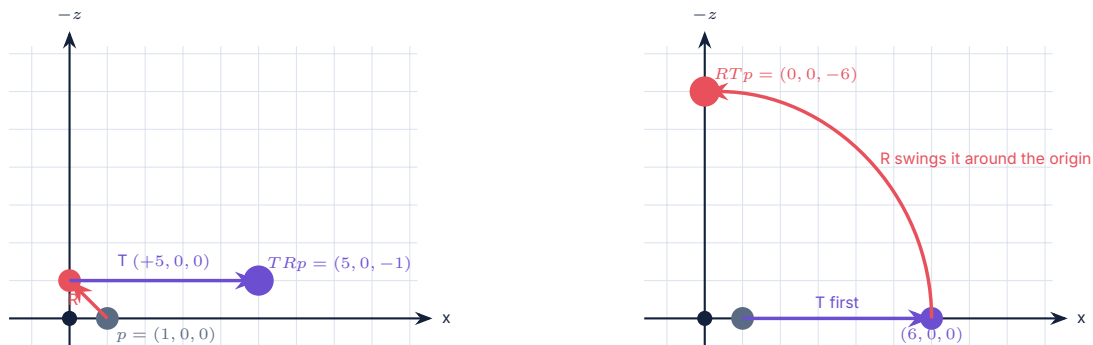
3.3 The transforms you need

Transform	Matrix	Effect
Translate	$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$	moves points by $\vec{t}$; directions unchanged
Scale	$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	stretches along the axes (negative = mirror)
Rotate about Y	$\begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$c = \cos \theta$, $s = \sin \theta$; counter-clockwise seen from $+Y$
Rotate about Z	$\begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	counter-clockwise seen from $+Z$

3.4 Order matters

Matrix multiplication is **not commutative**: $AB \neq BA$. With column vectors, $(TR)\vec{p}$ means apply R first, then T —read the product *right to left*. The listing below proves it with real numbers, and the figure shows the same result.

ROTATE-THEN-TRANSLATE VERSUS TRANSLATE-THEN-ROTATE (VIEWED FROM ABOVE: x RIGHT, $-z$ UP)



Rule of thumb: scene hierarchies • A model matrix is almost always $M = T \cdot R \cdot S$: scale first (in the object's own space), rotate, then place it in the world. A child object inherits its parent:

$M_{\text{world}} = M_{\text{parent}} \cdot M_{\text{local}}$. A wheel on a car, a hand on an arm: multiply down the tree.

3.5 Matrices in C++23

C++23 lets `operator[]` take *several* arguments, so a matrix can be indexed as `M[row, col]`, matching the math exactly. Storage is column-major, like Metal's `float4x4`.

gfx_mat.h: 4×4 matrices, lookAt, perspective

✓ header · compiled & used by the demo that follows

```
// gfx_mat.h -- 4x4 column-major matrices, Metal conventions
#pragma once
#include <array>
#include "gfx_vec.h"

struct Vec4 { float x = 0, y = 0, z = 0, w = 0; };

struct Mat4 {
    std::array<float, 16> d{}; // column-major: d[col * 4 + row]
    constexpr float& operator[](int row, int col) { return d[col * 4 + row]; }
    constexpr const float& operator[](int row, int col) const { return d[col * 4 + row]; }
    static constexpr Mat4 identity() {
        Mat4 m;
        for (int i = 0; i < 4; ++i) m[i, i] = 1.0f;
        return m;
    }
}
```

```

};

constexpr Mat4 operator*(const Mat4& a, const Mat4& b) {
    Mat4 r;
    for (int row = 0; row < 4; ++row)
        for (int col = 0; col < 4; ++col)
            for (int k = 0; k < 4; ++k)
                r[row, col] += a[row, k] * b[k, col];
    return r;
}

constexpr Vec4 operator*(const Mat4& m, Vec4 v) {
    return {m[0,0]*v.x + m[0,1]*v.y + m[0,2]*v.z + m[0,3]*v.w,
            m[1,0]*v.x + m[1,1]*v.y + m[1,2]*v.z + m[1,3]*v.w,
            m[2,0]*v.x + m[2,1]*v.y + m[2,2]*v.z + m[2,3]*v.w,
            m[3,0]*v.x + m[3,1]*v.y + m[3,2]*v.z + m[3,3]*v.w};
}

constexpr Mat4 translate(Vec3 t) {
    Mat4 m = Mat4::identity();
    m[0, 3] = t.x; m[1, 3] = t.y; m[2, 3] = t.z;
    return m;
}

constexpr Mat4 scale(Vec3 s) {
    Mat4 m = Mat4::identity();
    m[0, 0] = s.x; m[1, 1] = s.y; m[2, 2] = s.z;
    return m;
}

inline Mat4 rotateY(float radians) { // counter-clockwise seen from +Y
    Mat4 m = Mat4::identity();
    float c = std::cos(radians), s = std::sin(radians);
    m[0, 0] = c; m[0, 2] = s;
    m[2, 0] = -s; m[2, 2] = c;
    return m;
}

// View matrix: camera at `eye` looking at `target`; view space looks down -Z.
inline Mat4 lookAt(Vec3 eye, Vec3 target, Vec3 up) {
    Vec3 f = normalize(target - eye); // forward
    Vec3 r = normalize(cross(f, up)); // right
    Vec3 u = cross(r, f); // true up
    Mat4 m = Mat4::identity();
    m[0, 0] = r.x; m[0, 1] = r.y; m[0, 2] = r.z; m[0, 3] = -dot(r, eye);
    m[1, 0] = u.x; m[1, 1] = u.y; m[1, 2] = u.z; m[1, 3] = -dot(u, eye);
    m[2, 0] = -f.x; m[2, 1] = -f.y; m[2, 2] = -f.z; m[2, 3] = dot(f, eye);
    return m;
}

// Perspective projection, clip z in [0,1] after the divide (Metal/Direct3D style).
inline Mat4 perspective(float fovY, float aspect, float zNear, float zFar) {
    float ys = 1.0f / std::tan(fovY * 0.5f), xs = ys / aspect;
    Mat4 m;
    m[0, 0] = xs;
    m[1, 1] = ys;
    m[2, 2] = zFar / (zNear - zFar);
    m[2, 3] = zNear * zFar / (zNear - zFar);
    m[3, 2] = -1.0f; // w_clip = -z_view
    return m;
}

```

Order matters, directions ignore translation, and where M[1, 3] lives in memory

✓ compiled & run · g++ 13, -std=c++23 -Wall -Wextra -pedantic

```

#include <cstdio>
#include "gfx_mat.h"

float tidy(float v) { return std::fabs(v) < 1e-5f ? 0.0f : v; } // hide -0.0 from rounding

int main() {
    Vec4 p{1, 0, 0, 1}; // a point (w = 1)
    Vec4 d{1, 0, 0, 0}; // a direction (w = 0)
    Mat4 T = translate({5, 0, 0});
    Mat4 R = rotateY(std::numbers::pi_v<float> / 2); // 90 degrees
}

```

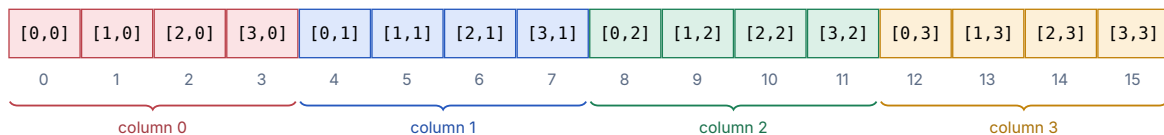
```
Vec4 a = (T * R) * p;           // rotate first, then translate
Vec4 b = (R * T) * p;           // translate first, then rotate
Vec4 c = T * d;                 // translation ignores directions
std::printf("T*R*p = (%.1f, %.1f, %.1f)\n", tidy(a.x), tidy(a.y), tidy(a.z));
std::printf("R*T*p = (%.1f, %.1f, %.1f)\n", tidy(b.x), tidy(b.y), tidy(b.z));
std::printf("T*d   = (%.1f, %.1f, %.1f, w=%.0f)\n", c.x, c.y, c.z, c.w);

Mat4 M{};
M[1, 3] = 7;                    // row 1, column 3
std::printf("storage index of M[1,3] = %d\n", int(&M[1, 3] - &M[0, 0]));
}
```

OUTPUT

```
T*R*p = (5.0, 0.0, -1.0)
R*T*p = (0.0, 0.0, -6.0)
T*d   = (1.0, 0.0, 0.0, w=0)
storage index of M[1,3] = 13
```

COLUMN-MAJOR STORAGE: 16 FLOATS IN MEMORY, MATRIX POSITION (row, col) $\mapsto 4 \cdot \text{col} + \text{row}$



Each group of four consecutive floats is one *column*. Upload a row-major matrix by mistake and every transform comes out transposed: objects rotate the wrong way and the translation lands in the bottom row. If a scene is mirrored or sheared, suspect this first.

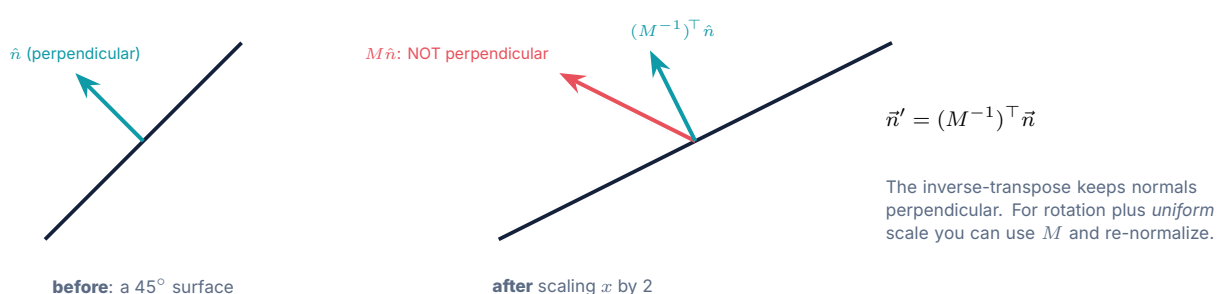
3.6 Rotations: Euler angles, axis-angle and quaternions

- **Euler angles** (yaw, pitch, roll) are easy to author but suffer *gimbal lock*: when the middle rotation reaches 90° two axes align and you lose a degree of freedom. Fine for an editor field, bad for animation.
- **Axis-angle**: rotate by θ about a unit axis $\hat{k}$. Rodrigues' formula: $\vec{v}' = \vec{v} \cos \theta + (\hat{k} \times \vec{v}) \sin \theta + \hat{k}(\hat{k} \cdot \vec{v})(1 - \cos \theta)$.
- **Quaternions** $q = (\hat{k} \sin \frac{\theta}{2}, \cos \frac{\theta}{2})$ store the same rotation in four numbers, compose with a cheap multiply, never gimbal-lock, and interpolate smoothly with *slerp*. Store orientations as quaternions; convert to a matrix once when building the model matrix.

3.7 Transforming normals: the inverse-transpose

Positions transform with M , but **normals do not**. Squash an object non-uniformly and a surface's tilt changes in the opposite sense; if you push normals through the same matrix they stop being perpendicular to the surface.

NON-UNIFORM SCALE BREAKS NAIVE NORMALS



Pitfall: negative scale flips winding • A mirror ($s_x = -1$) has a negative determinant: it turns a counter-clockwise triangle clockwise, so back-face culling removes the *front* faces. Detect $\det M_{3 \times 3} < 0$ on the CPU and flip the cull mode (or the winding) for that draw.

Chapter 3 in one minute

- Each matrix column is where an axis lands. A 4×4 with $w = 1$ for points / $w = 0$ for directions does rotate, scale and translate together.
- With column vectors, ABp applies B first. Compose models as $T R S$ and parents as $M_{\text{parent}} M_{\text{local}}$.
- Metal and this guide are column-major; C++23 gives you `M[row, col]`.
- Normals need $(M^{-1})^\top$; mirrors flip winding; store rotations as quaternions.

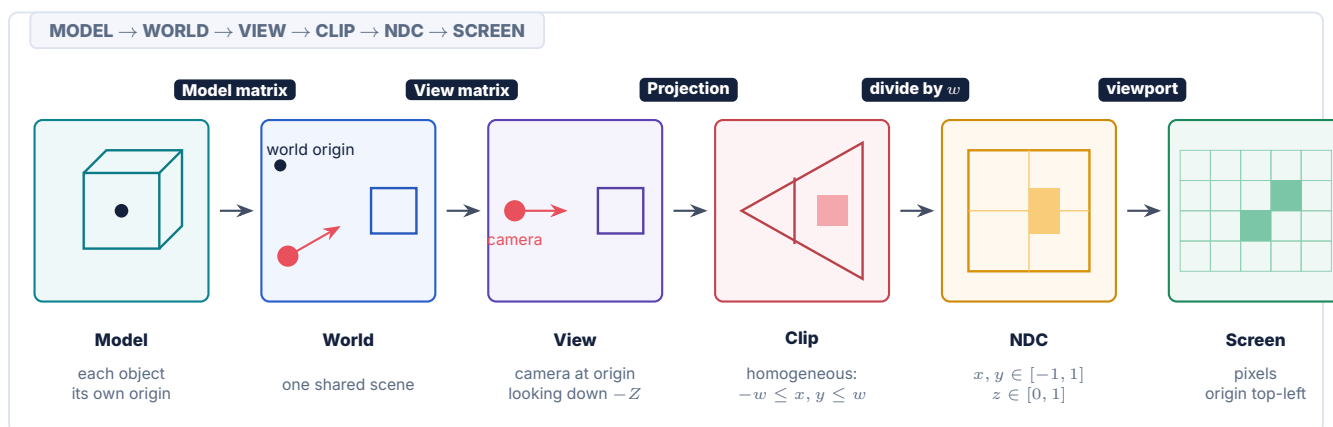
Cameras and Projection

How a point in your 3D world travels through five coordinate spaces to land on a pixel: the view matrix, perspective, clip space, the depth buffer's precision problem, and reversed-Z.

04

4.1 The journey of a vertex

A vertex starts life in the **model space** of the object it belongs to and ends as a pixel position plus a depth. Between them it passes through a fixed chain of spaces; each hop is one matrix multiply, except the last two.



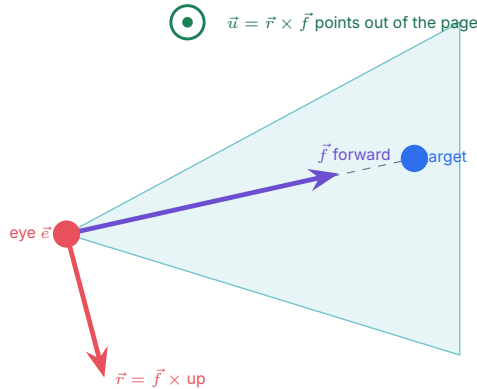
In a vertex shader you normally combine the first three into one matrix on the CPU, $MVP = PVM$, and the shader does a single multiply: `out.position = mvp * float4(pos, 1)`. The hardware performs the last two steps itself.

4.2 The view matrix

The camera is just an object with a position and orientation. Instead of moving the camera we move *the whole world* the opposite way, so the camera ends up at the origin looking down $-Z$. That means the view matrix is the **inverse of the camera's own transform**.

`LookAt` builds it from three things: where the camera is (*eye*), what it looks at (*target*) and which way is roughly up. It constructs the camera's three axes, then uses the fact that for perpendicular unit axes the inverse of a rotation is its transpose.

BUILDING THE CAMERA FRAME FROM lookAt(eye, target, up) (VIEWED FROM ABOVE)



$$V = \begin{pmatrix} r_x & r_y & r_z & -\vec{r} \cdot \vec{e} \\ u_x & u_y & u_z & -\vec{u} \cdot \vec{e} \\ -f_x & -f_y & -f_z & \vec{f} \cdot \vec{e} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rows are the camera's axes (a rotation whose transpose is its inverse). The last column is $-\vec{R}\vec{e}$: move the eye to the origin first. The $-\vec{f}$ row is why view space looks down $-Z$.

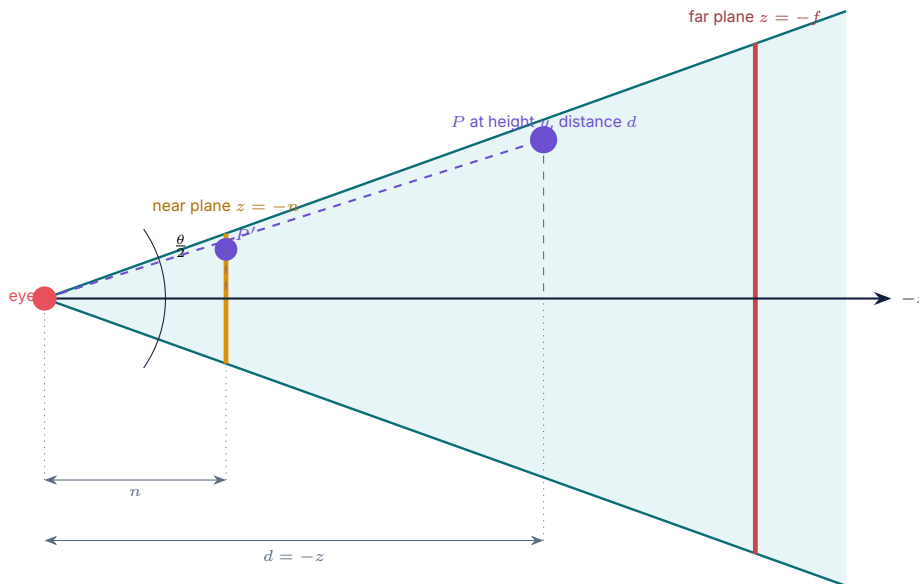
4.3 Perspective: similar triangles

Things look smaller with distance because the rays through your eye form similar triangles. A point at height y and distance $d = -z$ lands on a screen placed at distance n at height

$$y' = y \cdot \frac{n}{d} \implies \text{divide by the distance.}$$

So perspective is, at heart, a division by depth. Matrices cannot divide, but they *can* copy $-z$ into the fourth component w ; then the hardware divides x, y, z by w after the vertex shader (the **perspective divide**). That is the entire trick.

SIDE VIEW OF THE FRUSTUM: THE FIELD OF VIEW AND SIMILAR TRIANGLES



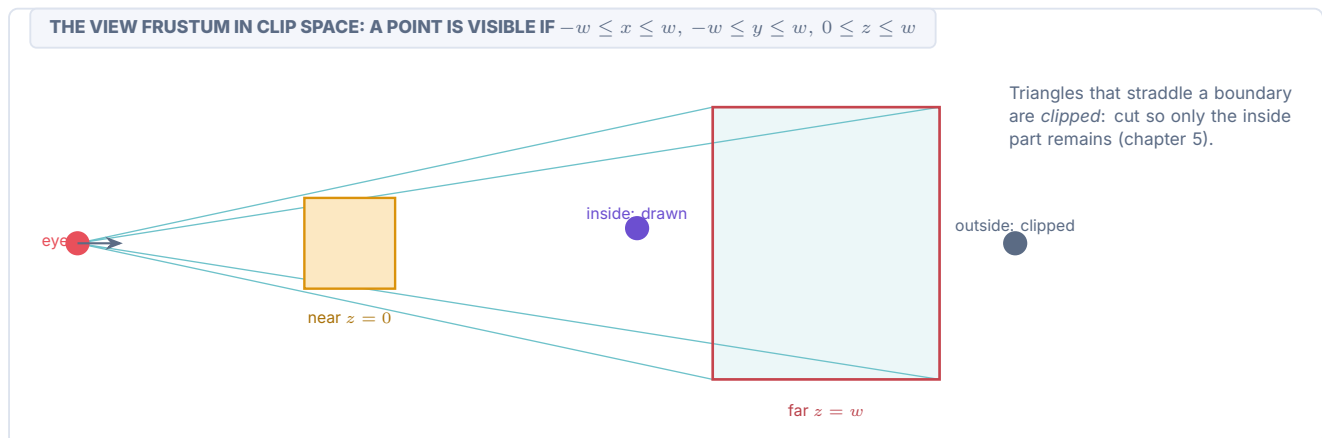
Both small and large triangles share the same angle at the eye, so $y'/n = y/d$, which gives $y' = y n/d$.

Metal's perspective matrix for vertical field of view θ , aspect ratio $a = \text{width}/\text{height}$ and clip distances n, f is

$$P = \begin{pmatrix} \frac{1}{a \tan \frac{\theta}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan \frac{\theta}{2}} & 0 & 0 \\ 0 & 0 & \frac{f}{n-f} & \frac{nf}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

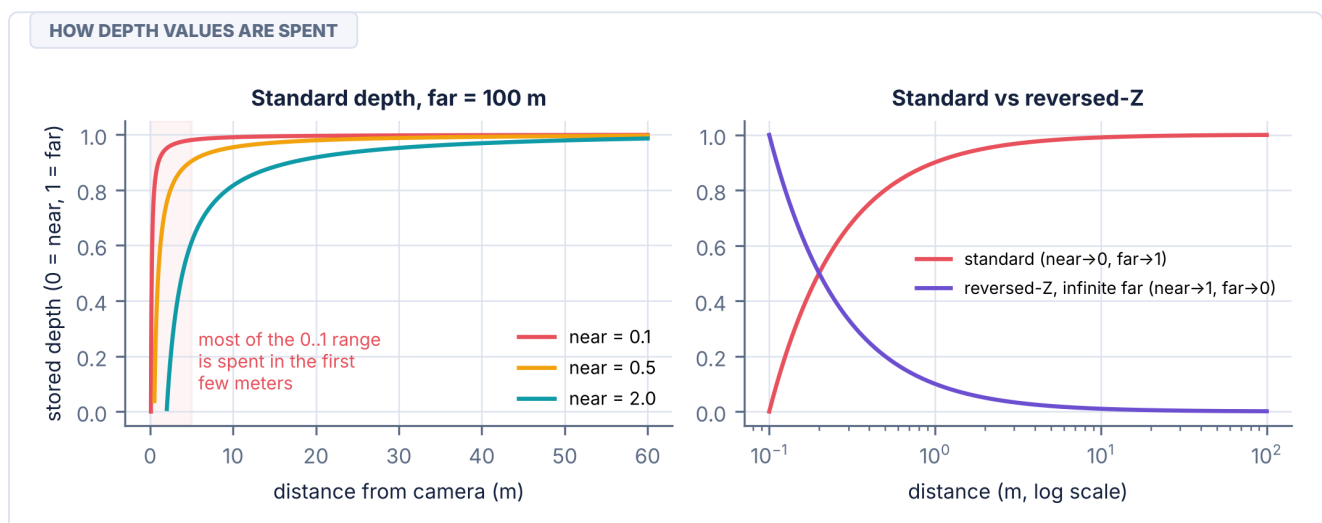
The bottom row $(0, 0, -1, 0)$ is the divide-by-depth trick: $w_{\text{clip}} = -z_{\text{view}}$. The third row maps view-space depth

into $[0, 1]$: $z_{\text{ndc}} = \frac{f}{f-n} \left(1 - \frac{n}{d}\right)$ with $d = -z_{\text{view}}$, which is 0 at the near plane and 1 at the far plane. (OpenGL's matrix maps to $[-1, 1]$ instead; do not mix them.)



4.4 Depth precision, and why you want reversed-Z

The depth buffer does not store distance; it stores z_{ndc} , which is a function of $1/d$. That is wonderful for perspective (it stays a linear function across a triangle in screen space) but terrible for precision: nearly the whole $[0, 1]$ range is used up in the first few meters and far objects share just a few representable values, so distant surfaces z-fight.



Reversed-Z flips the mapping: near $\rightarrow 1$, far $\rightarrow 0$. Floating-point numbers are densest near 0, so the little precision hyperbolic depth leaves for distant surfaces is paired with the finest floats, and the two effects almost cancel. With an *infinite* far plane the matrix even simplifies: set $m[2, 2] = 0$, $m[2, 3] = n$, $m[3, 2] = -1$, giving $z_{\text{ndc}} = n/d$. The program below measures it: for a surface 0.1% farther away, how many distinct 32-bit float depth values separate the two?

Counting distinct depth values: standard versus reversed-Z

✓ compiled & run · g++13, -std=c++23 -Wall -Wextra -pedantic

```
#include <bit>
#include <cmath>
#include <stdint>
#include <stdio>
#include <stdlib>
#include <initializer_list>

// Depth for a point at view distance d (positive), near plane n, far plane f.
float depthStandard(float d, float n, float f) { return f * (d - n) / (d * (f - n)); } // near 0, far 1
```

```
float depthReversed(float d, float n) { return n / d; } // near 1, far 0

// How many distinct floats lie between two depths? (floats order like their bit patterns)
int64_t ulpsBetween(float a, float b) {
    return std::abs(int64_t(std::bit_cast<int32_t>(a)) - int64_t(std::bit_cast<int32_t>(b)));
}

int main() {
    const float n = 0.1f, f = 1000.0f;
    for (float d : {5.0f, 50.0f, 500.0f}) {
        float d2 = d * 1.001f; // a surface 0.1% farther away
        std::printf("distance %5.0f: standard %6lld ULPs apart, reversed-Z %6lld ULPs apart\n", d,
                    (long long)ulpsBetween(depthStandard(d, n, f), depthStandard(d2, n, f)),
                    (long long)ulpsBetween(depthReversed(d, n), depthReversed(d2, n)));
    }
}
```

OUTPUT

```
distance    5: standard    335 ULPs apart, reversed-Z 10727 ULPs apart
distance   50: standard     33 ULPs apart, reversed-Z  8582 ULPs apart
distance  500: standard      3 ULPs apart, reversed-Z 13731 ULPs apart
```

At a distance of 500 m the standard mapping separates two surfaces by only **3** representable depth values, so they z-fight; reversed-Z gives over **13 000**. To use it in Metal you must (a) build the reversed matrix, (b) clear depth to **0** instead of 1, and (c) set the depth compare to Greater (chapter 10 shows the descriptor).

Use a Depth32Float format • Reversed-Z only pays off with a floating-point depth buffer. A 24-bit integer depth format gets no benefit, and Apple GPUs' native depth format is 32-bit float anyway.

4.5 A vertex's whole trip, with numbers

This program pushes one vertex through every stage with the library from chapter 3 and prints each result. Follow the numbers against the figure at the top of the chapter.

From model space to a pixel, step by step

✓ compiled & run • g++13, -std=c++23 -Wall -Wextra -pedantic

```
#include <cstdio>
#include "gfx_mat.h"

int main() {
    const int W = 1280, H = 720;
    Mat4 model = translate({0, 0, -2}) * rotateY(0.0f); // object 2 units from origin
    Mat4 view = lookAt({0, 0, 3}, {0, 0, 0}, {0, 1, 0}); // camera at z=3, facing -z
    Mat4 proj = perspective(std::numbers::pi_v<float> / 3, float(W) / H, 0.1f, 100.0f);

    Vec4 local{0.5f, 0.5f, 0.0f, 1.0f}; // a vertex in model space
    Vec4 world = model * local;
    Vec4 eye = view * world;
    Vec4 clip = proj * eye;
    std::printf("world = (%.3f, %.3f, %.3f)\n", world.x, world.y, world.z);
    std::printf("view = (%.3f, %.3f, %.3f) <- looks down -z\n", eye.x, eye.y, eye.z);
    std::printf("clip = (%.3f, %.3f, %.3f, w=%.3f)\n", clip.x, clip.y, clip.z, clip.w);

    float ndcX = clip.x / clip.w, ndcY = clip.y / clip.w, ndcZ = clip.z / clip.w;
    std::printf("ndc = (%.3f, %.3f, %.3f) <- z in [0,1]\n", ndcX, ndcY, ndcZ);

    float px = (ndcX * 0.5f + 0.5f) * W; // NDC x: -1..1 -> 0..W
    float py = (1.0f - (ndcY * 0.5f + 0.5f)) * H; // NDC y up, pixel y down
    std::printf("pixel = (%.1f, %.1f) depth = %.5f\n", px, py, ndcZ);

    for (float z : {-0.1f, -1.0f, -10.0f, -100.0f}) { // depth vs distance
        Vec4 c = proj * Vec4{0, 0, z, 1};
        std::printf("z_view = %7.1f -> depth %.5f\n", z, c.z / c.w);
    }
}
```

OUTPUT

```
world = ( 0.500, 0.500, -2.000)
view = ( 0.500, 0.500, -5.000) <- looks down -z
clip = ( 0.487, 0.866, 4.905, w= 5.000)
ndc = ( 0.097, 0.173, 0.981) <- z in [0,1]
pixel = (702.4, 297.6) depth = 0.98098
z_view = -0.1 -> depth 0.00000
z_view = -1.0 -> depth 0.90090
z_view = -10.0 -> depth 0.99099
z_view = -100.0 -> depth 1.00000
```

Read the output top to bottom: the camera is at $z = 3$ and the object at $z = -2$, so the vertex is 5 units in front of the camera ($z_{\text{view}} = -5$), and that same 5 appears as w in clip space. Dividing by it gives NDC; the viewport mapping then gives pixel (702.4, 297.6). The last four lines show how depth bunches up: at 1m already 0.90, at 10 m 0.99.

4.6 A camera you can fly

Games rarely call `lookAt` with hand-picked targets. A first-person or editor camera stores a position plus **yaw** (turn left/right) and **pitch** (look up/down), derives the forward vector from them, and rebuilds the view matrix each frame. The listing verifies itself: a point placed 3 m straight ahead must land at (0, 0, -3) in view space.

A yaw/pitch fly camera built on `lookAt`

✓ compiled & run · g++ 13, -std=c++23 -Wall -Wextra -pedantic

```
#include <algorithm>
#include <cstdio>
#include "gfx_mat.h"

struct FlyCamera {
    Vec3 position{0, 1.7f, 4};
    float yaw = 0, pitch = 0; // radians; yaw 0 looks down -z

    Vec3 forward() const {
        return {std::cos(pitch) * std::sin(yaw),
                std::sin(pitch),
                -std::cos(pitch) * std::cos(yaw)};
    }
    Vec3 right() const { return normalize(cross(forward(), {0, 1, 0})); }

    void look(float dYaw, float dPitch) { // mouse deltas, already in radians
        yaw += dYaw;
        constexpr float limit = 89.0f * std::numbers::pi_v<float> / 180.0f;
        pitch = std::clamp(pitch + dPitch, -limit, limit); // never reach straight up: lookAt breaks
    }
    void move(float ahead, float strafe) { position += forward() * ahead + right() * strafe; }
    Mat4 view() const { return lookAt(position, position + forward(), {0, 1, 0}); }
};

int main() {
    FlyCamera cam;
    cam.look(0.6f, -0.2f); // turn right and look slightly down
    cam.move(2.0f, 0.5f);

    Vec3 target = cam.position + cam.forward() * 3.0f; // a point 3 m straight ahead
    Vec4 v = cam.view() * Vec4{target.x, target.y, target.z, 1};
    std::printf("view-space position of that point: (%.3f, %.3f, %.3f)\n", v.x, v.y, v.z);
    std::printf("expected: (0, 0, -3): on the view axis, 3 m in front\n");
}
```

OUTPUT

```
view-space position of that point: (0.000, -0.000, -3.000)
expected: (0, 0, -3): on the view axis, 3 m in front
```

Pitfall: looking straight up • When forward becomes parallel to the up vector the cross product $\vec{f} \times \text{up}$ collapses to zero and the whole matrix turns to NaN. Clamp pitch just short of $\pm 90^\circ$, as above, or switch to a quaternion-based camera.

4.7 Orthographic projection and the viewport

An **orthographic** camera keeps parallel lines parallel: no divide by depth, w stays 1. It is what you want for shadow-map cascades, UI, CAD views and isometric games. For a box $[l, r] \times [b, t] \times [n, f]$:

$$P_{\text{ortho}} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{1}{n-f} & \frac{n}{n-f} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The **viewport transform** maps NDC to pixels, flipping y because NDC points up but framebuffers grow down:

$$x_{\text{px}} = \left(\frac{x_{\text{ndc}} + 1}{2} \right) W, \quad y_{\text{px}} = \left(\frac{1 - y_{\text{ndc}}}{2} \right) H$$

Pixel *centers* sit at half-integers: pixel $(0, 0)$ covers $[0, 1)^2$ and its center is $(0.5, 0.5)$. Rasterization tests that center point (chapter 5), and temporal anti-aliasing nudges the projection by sub-pixel *jitter* each frame to gather more samples.

Cheat sheet: field of view

$$\text{hfov} = 2 \arctan\left(a \tan \frac{\text{vfov}}{2}\right)$$

convert vertical to horizontal; $a = W/H$

60° vertical at 16:9 $\approx$ 91° horizontal

a common game default

Larger FOV $\Rightarrow$ more of the scene, stronger perspective distortion at the edges

sprinting and speed effects zoom" the FOV

Near plane: as large as you can get away with (0.05–0.3 m)

depth precision scales with f/n

Chapter 4 in one minute

- Vertices go model $\rightarrow$ world $\rightarrow$ view $\rightarrow$ clip (matrices), then $\div w$ gives NDC and the viewport gives pixels (hardware).
- The view matrix is the inverse of the camera transform; `lookAt` builds it from eye, target, up.
- Perspective divides by depth by copying $-z$ into w . Visible means $-w \leq x, y \leq w$ and $0 \leq z \leq w$.
- Stored depth is nonlinear in distance; use reversed-Z with a float depth buffer (clear 0, compare Greater).

Rasterization

How the hardware decides which pixels a triangle covers, interpolates data across it, handles perspective, clipping, depth and anti-aliasing. Every picture here comes from a software rasterizer written for this book.

05

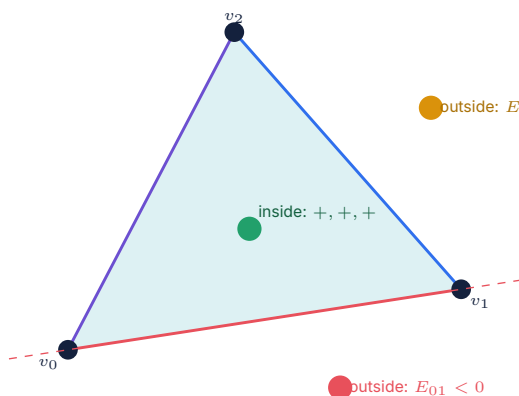
5.1 Which pixels does a triangle cover?

After the vertex shader, a triangle is just three 2D points on screen. The rasterizer must decide, for each pixel, whether the triangle covers it. The tool is the **edge function**: for an edge from A to B and any point P ,

$$E_{AB}(P) = (B_x - A_x)(P_y - A_y) - (B_y - A_y)(P_x - A_x)$$

This is the 2D cross product of $B - A$ and $P - A$. It is positive when P is on one side of the line through A and B , negative on the other, and zero on the line. A point is inside the triangle when it is on the *same* side of all three edges.

THE EDGE FUNCTION: WHICH SIDE OF AN EDGE IS THE PIXEL ON?



Test: compute E_{01}, E_{12}, E_{20} at the pixel center. If all are ≥ 0 the pixel is covered (for counter-clockwise triangles on a y -up grid; flip the signs for the other winding).

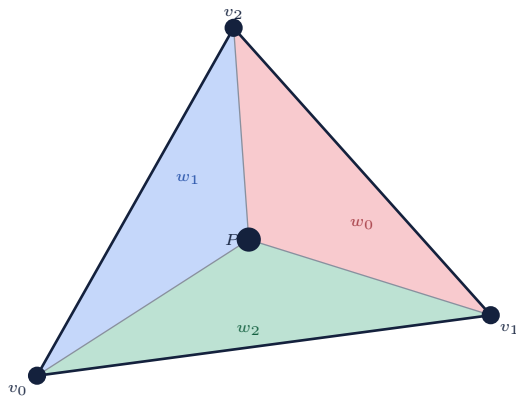
The sign of the triangle's own area, $E_{01}(v_2)$, is its winding order: it tells you the facing for back-face culling.

Edge functions are linear in x and y , so a GPU can step them with additions and test a whole tile of pixels in parallel.

5.2 Barycentric coordinates

Divide the three edge-function values by the triangle's total area and you get three weights (w_0, w_1, w_2) that sum to 1. These are the **barycentric coordinates** of the pixel: how much of each vertex the pixel is made of". Geometrically, each weight is the *area of the sub-triangle opposite that vertex* as a fraction of the whole.

BARYCENTRIC WEIGHTS ARE SUB-AREAS, AND THEY BLEND ANY VERTEX DATA



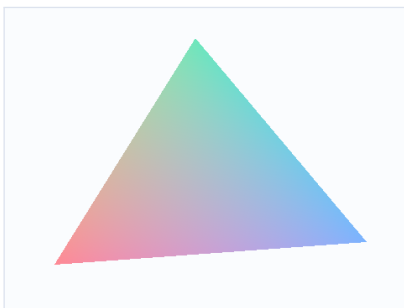
$$w_0 = \frac{\text{area}(Pv_1v_2)}{\text{area}(v_0v_1v_2)} = \frac{E_{12}(P)}{E_{12}(v_0)}$$

$$w_0 + w_1 + w_2 = 1, \quad P = w_0v_0 + w_1v_1 + w_2v_2$$

$$\text{value}(P) = w_0 a_0 + w_1 a_1 + w_2 a_2$$

The same weights blend *any* per-vertex quantity (a): color, normal, UV, depth. Inside the triangle all weights are ≥ 0 .

BARYCENTRIC INTERPOLATION OF THREE VERTEX COLORS (RENDERED BY THE LISTING BELOW'S ALGORITHM)



each pixel's color = $w_0c_0 + w_1c_1 + w_2c_2$

Red at one corner, green at another, blue at the third: every pixel in between is a weighted mix. This is exactly what the rasterizer hands your fragment shader as `[[stage_in]]` data. Normals, texture coordinates and anything else you write from the vertex shader are interpolated the same way.

A complete edge-function rasterizer with a depth buffer (ASCII output)

✓ compiled & run · g++ 13, -std=c++23 -Wall -Wextra -pedantic

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <vector>

struct P {
    float x, y;
};

// Edge function: positive when c lies to the left of the directed edge a->b.
// It is twice the signed area of triangle (a, b, c).
float edge(P a, P b, P c) { return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x); }

int main() {
    const int W = 36, H = 16;
    P v0{3, 2}, v1{33, 7}, v2{14, 14}; // screen-space triangle (y grows downward)
    float area = edge(v0, v1, v2); // sign = winding, magnitude = 2 * area
    std::vector<float> depth(W * H, 1.0f); // depth buffer, cleared to the far value
    float z0 = 0.2f, z1 = 0.9f, z2 = 0.5f; // per-vertex depth

    for (int y = 0; y < H; ++y) {
        for (int x = 0; x < W; ++x) {
            P p{x + 0.5f, y + 0.5f}; // sample at the pixel center
            float w0 = edge(v1, v2, p) / area; // barycentric weight of v0
            float w1 = edge(v2, v0, p) / area; // ... of v1
            float w2 = edge(v0, v1, p) / area; // ... of v2
            bool inside = w0 >= 0 && w1 >= 0 && w2 >= 0;
            char c = '.';
        }
    }
}
```

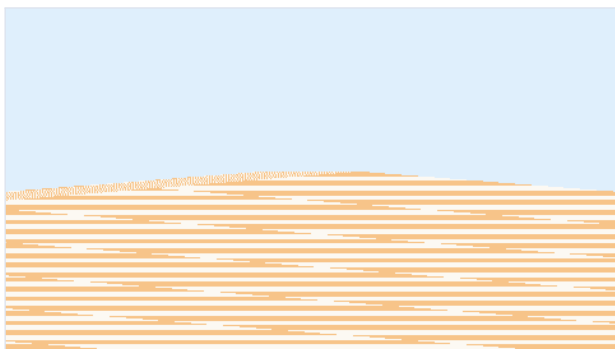


```
std::printf(" %8.1f | %6.3f | %6.3f\n", sx, naive, correct);
}
```

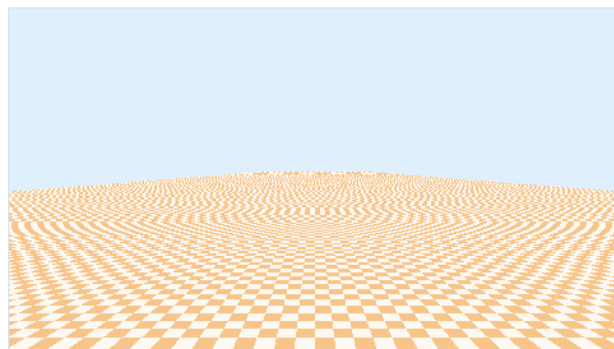
OUTPUT

screen x	naive u	perspective-correct u
0.0	0.000	0.000
25.0	0.250	0.036
50.0	0.500	0.100
75.0	0.750	0.250
100.0	1.000	1.000

THE SAME CHECKERBOARD FLOOR: LEFT, ONE HUGE QUAD INTERPOLATED IN SCREEN SPACE; RIGHT, PERSPECTIVE-CORRECT



screen-space (affine) interpolation: wrong

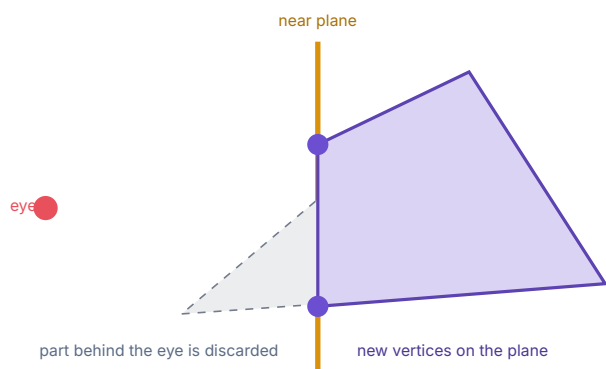


perspective-correct interpolation

5.4 Clipping

Triangles that cross the edges of the view volume must be cut. The one that cannot be skipped is the **near plane**: a vertex with $w \leq 0$ is at or behind the eye, and dividing by w would flip it to the wrong side of the screen. The hardware clips the triangle against the planes in clip space, producing one or two new triangles with new vertices whose attributes are interpolated along the cut edge.

CLIPPING A TRIANGLE THAT CROSSES THE NEAR PLANE (SIDE VIEW)



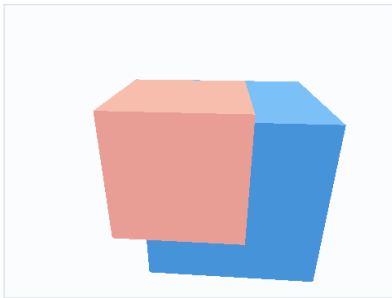
One vertex was in front of the plane, two behind (or the reverse). The clipper replaces the triangle with one or two triangles whose new vertices lie exactly on the plane, with interpolated attributes.

Left, right, top and bottom edges are usually *not* clipped: a guard band lets the rasterizer simply skip pixels outside the viewport.

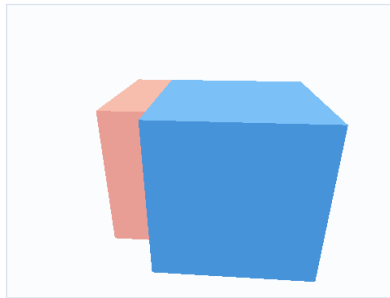
5.5 The depth buffer

When triangles overlap on screen, which is in front? The **painter's algorithm** (draw far to near) fails for intersecting or cyclic shapes and costs a sort. The **depth buffer** solves it per pixel: it stores the nearest depth seen so far at each pixel; a new fragment is kept only if its depth passes the test (smaller wins with the usual convention, larger with reversed-Z), and then replaces the stored value.

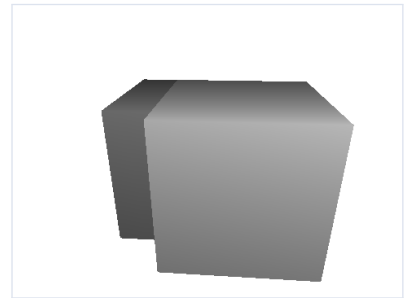
THE SAME TWO OVERLAPPING CUBES: WITHOUT AND WITH A DEPTH TEST



no depth test: whichever draws last wins
(wrong)



depth test: nearest surface wins



the depth buffer itself (dark = near)

On Apple GPUs • Because of tile-based deferred rendering, opaque overdraw is resolved by hardware before the fragment shader runs, so heavy fragment shaders on hidden surfaces are usually skipped. Things that defeat this: `discard` and alpha-tested geometry, writing depth from the shader, and blending (translucent surfaces must be drawn after opaques, back to front).

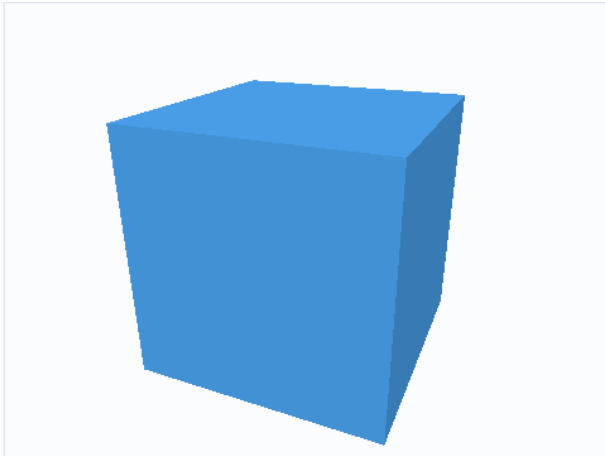
Depth bias

Shadow maps and decals compare depths of surfaces that are nearly coplanar, so floating-point error makes them flicker (shadow acne, z-fighting). A small **depth bias** (constant plus slope-scaled) nudges stored depth to break ties; Metal exposes it as `setDepthBias(constant, slopeScale, clamp)`.

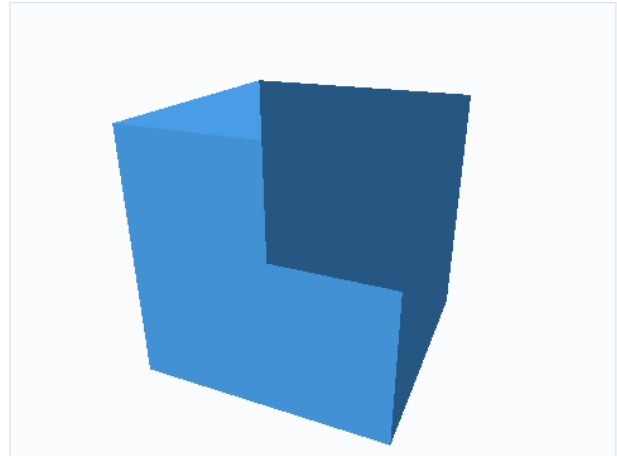
5.6 Back-face culling and winding

A closed object's far side can never be visible, so the GPU discards triangles that face away. Facing is decided by the sign of the projected triangle area, i.e. the vertex order (**winding**). Pick a convention—here counter-clockwise from outside is front—and tell the API with `setFrontFacingWinding`. Culling roughly halves the triangles to shade.

A CUBE WITH CULLING (LEFT) AND WITH CULLING AND DEPTH BOTH OFF (RIGHT): INSIDE FACES LEAK THROUGH



back faces culled



no culling, no depth test: back faces painted over front

Pitfall: the invisible mesh • If everything vanishes after importing a model, the winding is probably the opposite of your convention (or a mirror matrix flipped it, chapter 3). Flip the winding state or the cull mode before touching any math.

5.7 Anti-aliasing: coverage at sub-pixel level

A pixel is either inside a triangle or not, so edges become staircases (*aliasing*). **MSAA** (multisample anti-aliasing) tests coverage at several sample points per pixel, shades *once* per pixel, and the fraction of covered samples blends the edge during the resolve. The pictures below zoom a triangle corner; edges are at identical positions in all three.

ONE, FOUR AND SIXTEEN COVERAGE SAMPLES PER PIXEL (12× ZOOM OF A TRIANGLE CORNER)



1 sample: binary edge



4 samples: 5 levels of gray



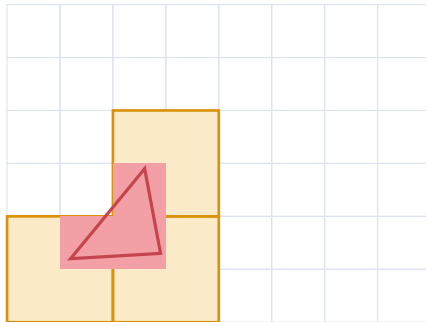
16 samples: 17 levels

MSAA only smooths *geometric* edges, not shader aliasing (sparkly specular highlights, alpha-tested foliage). It is cheap on Apple GPUs because the samples live in tile memory. Modern engines often use **temporal anti-aliasing** (accumulate jittered frames, chapter 4) or MetalFX upscalers instead, especially with deferred or ray-traced shading where MSAA is awkward.

5.8 Quads, derivatives and the cost of small triangles

Fragment shaders run in groups of 2×2 pixels called **quads**, so the GPU can compare neighbors and measure how fast the texture coordinate changes across a pixel (its *derivatives*). Texture sampling uses them to choose a mip level (chapter 6). The price: even a triangle covering a single pixel launches four shader invocations, and three are discarded *helper lanes*.

QUAD OVERSHADING: A SLIVER TRIANGLE COVERS 3 PIXELS BUT COSTS 12 FRAGMENT INVOCATIONS



Red: pixels the triangle really covers.

Amber: the 2×2 quads the GPU must run to cover them.

Triangles smaller than a few pixels waste most of their work on helper lanes. This is why very dense meshes shade slowly, and why engines render tiny geometry with different techniques (visibility buffers, mesh-shader culling, LODs).

Cheat sheet: rasterization rules of thumb

Sample at pixel centers ($x + 0.5, y + 0.5$)	matches Metal's rasterization rule
Top-left fill rule	a pixel exactly on a shared edge belongs to exactly one of the two triangles, so no gaps and no double-blend
Interpolate u/w and $1/w$	the hardware does this for you; software rasterizers must
Cull back faces for closed meshes	flip winding after mirror transforms
Opaque front-to-back, translucent back-to-front	helps early rejection and blending correctness

Chapter 5 in one minute

- Edge functions say which side of an edge a pixel is on; inside means all three agree. Normalized, they are barycentric weights that blend any vertex data.
- Interpolation must be perspective-correct (u/w and $1/w$). Near-plane clipping creates new vertices; the rest is guard-banded.
- The depth buffer keeps the nearest fragment per pixel; cull back faces by winding; MSAA tests coverage at sub-pixel samples.
- Shading happens in 2×2 quads, so tiny triangles are expensive.

Textures and Sampling

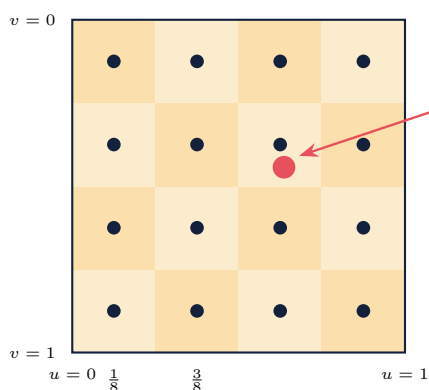
Images on the GPU are not arrays you index; they are sampled, filtered and compressed by dedicated hardware. This chapter shows what that hardware does, why mipmaps exist, and how to create and bind a texture in Metal 4.

6.1 UV coordinates and texel centers

A texture is a grid of **texels** (texture pixels). You do not address them by integer index in a shader; you give the sampler a **UV coordinate** in $[0, 1]^2$ that is independent of the texture's resolution, so the same shader works for a 256^2 and a 4096^2 image. By the convention of this guide, $(0, 0)$ is the *top-left* corner of the image and v grows downward.

The detail that trips everyone up: a texel's value belongs to its *center*, not its corner. In a 4×4 texture the first texel center is at $u = \frac{1}{8}$, not 0.

UV SPACE AND TEXEL CENTERS (A 4×4 TEXTURE)



Dots mark texel centers at $(i + \frac{1}{2})/N$. The value stored in a texel is the color at the dot.

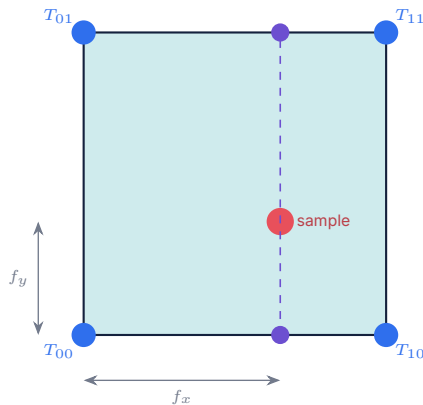
A sample at $(\frac{1}{2}, \frac{1}{2})$ falls on a corner shared by four texels, so with linear filtering it returns the average of all four.

Integer texel fetch: $\text{texel} = \lfloor uN \rfloor$. Half-texel offsets matter whenever you copy or reproject images.

6.2 Filtering when the texture is magnified

When a texture covers more screen pixels than it has texels, each sample falls *between* texel centers. The sampler has to invent a value. **Nearest** filtering takes the closest texel (blocky). **Linear** (bilinear) filtering blends the four surrounding texels, weighting each by how close the sample is.

BILINEAR FILTERING: THREE LERPS



$$a = \text{lerp}(T_{00}, T_{10}, f_x)$$

$$b = \text{lerp}(T_{01}, T_{11}, f_x)$$

$$\text{result} = \text{lerp}(a, b, f_y)$$

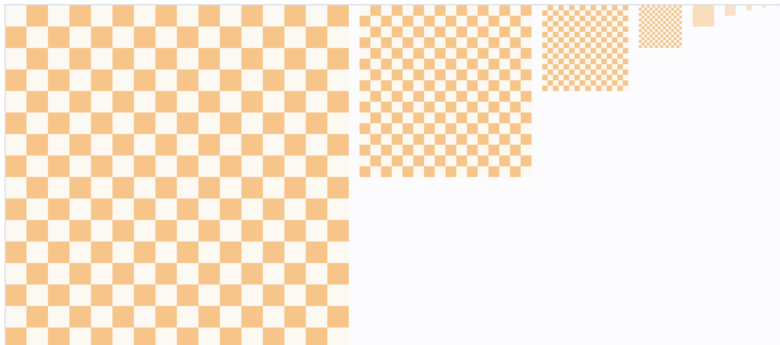
f_x, f_y are the fractional distances to the lower-left texel center. Dedicated hardware does this in one instruction per sample, which is why textures are the cheapest way to read smooth data, including non-image data such as lookup tables.

6.3 Minification and aliasing: mipmaps

The opposite problem is worse. A distant checkerboard may have a thousand texels under one pixel. Sampling one of them at random gives noise that shimmers when the camera moves. The correct answer is the *average* of everything under the pixel, but averaging a thousand texels per sample is far too slow.

The fix is to precompute it. A **mipmap chain** stores the texture and then successively halved copies, each the average of 2×2 texels of the one before, down to 1×1 . At draw time the hardware picks the level where one texel is about one pixel wide.

A MIP CHAIN: EACH LEVEL IS HALF THE SIZE, THE WHOLE CHAIN COSTS ONLY $\frac{4}{3}$ OF LEVEL 0



levels 0, 1, 2, 3, ... of a checkerboard (generated by averaging 2×2 blocks)

$$\text{memory} = 1 + \frac{1}{4} + \frac{1}{16} + \dots = \frac{4}{3}$$

So mipmaps cost 33% extra memory, and in return they remove shimmering, shrink the working set the cache has to hold, and often speed rendering up.

Number of levels: $\lfloor \log_2 \max(w, h) \rfloor + 1$.

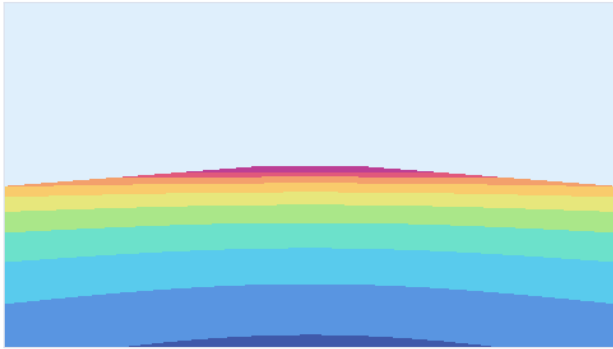
How the GPU picks a level

Remember the 2×2 quads from chapter 5: neighbors in a quad let the hardware measure how many texels the UV changes over, per pixel step. If the UV footprint of one pixel is ρ texels wide, the level of detail is

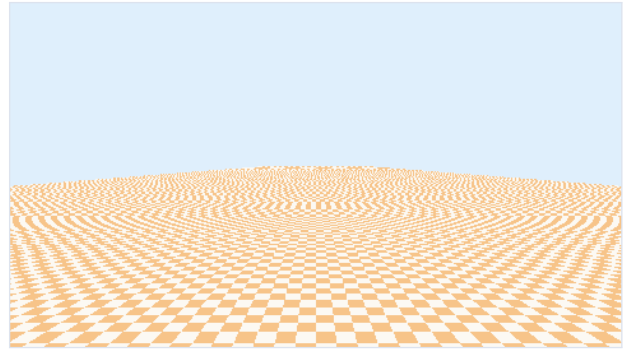
$$\lambda = \log_2 \rho$$

Level $\lambda = 0$ is the full-resolution image, $\lambda = 1$ is half size, and so on. Because the derivatives come from the quad, mip selection **only works in uniform control flow**: a `sample` call after a divergent `if` can read garbage derivatives. When you must sample in a branch, compute the level yourself and use `level(x)` or the gradient overload.

THE LEVEL CHOSEN FOR A FLOOR PLANE: BANDS OF CONSTANT MIP LEVEL (COLOR = LEVEL)



near the camera: level 0 (blue); farther: higher levels, each band twice as dense



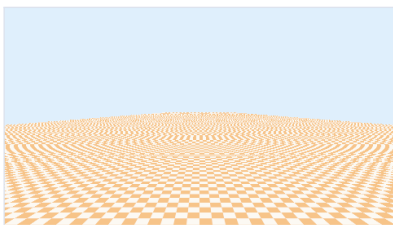
without mipmaps: distant texels alias into moiré

6.4 Trilinear and anisotropic filtering

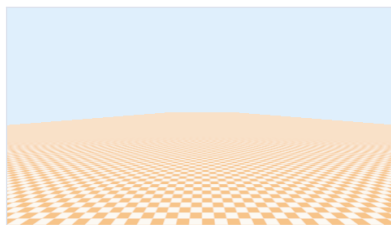
Choosing the single nearest level makes visible seams where the level changes. **Trilinear** filtering takes a bilinear sample from the two nearest levels and blends them by the fractional part of λ : eight texels per sample, no seams.

Trilinear still assumes the pixel footprint is square. On a floor seen at a grazing angle the footprint is a long thin sliver: many texels along the view direction, one across. Choosing the level by the long side keeps it sharp across but blurs along; choosing by the short side makes it sharp but aliasing. **Anisotropic filtering** takes several samples along the long axis of the footprint (up to a limit such as 16) and averages them.

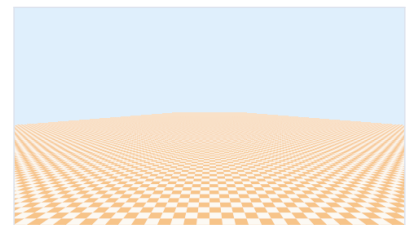
FILTER QUALITY ON A GRAZING FLOOR: BILINEAR, TRILINEAR, ANISOTROPIC



bilinear, no mips: shimmer in the distance



trilinear: stable but blurry toward the horizon



anisotropic: sharp and stable

Default recipe • For color textures: linear min/mag, linear mip filter, anisotropy 8–16, repeat addressing. For data you do not want blurred (IDs, lookup indices): nearest, no mips. On Apple GPUs anisotropy costs extra samples, so profile it, but 8× is almost always affordable.

6.5 Color spaces and pixel formats

Chapter 7 explains *why* images are stored with a gamma curve. What matters for texturing is a single rule: **filtering must happen on linear values**. Averaging sRGB-encoded numbers makes edges and mip levels too dark. Metal solves this for you if you pick the right format: an `_sRGB` format is decoded to linear *before* the filter runs, and encoded after a render-target write.

Which format for which data?

Content	Format	Why
Albedo / base color	RGBA8Unorm_sRGB BC7/ASTC sRGB)	(or Authored as sRGB; hardware linearizes before filtering
Normal map	RGBA8Unorm, BC5, ASTC	Vectors, not colors: never sRGB
Roughness / metalness / AO	R8Unorm or packed	Data. Linear
HDR render target	RGBA16Float	Values above 1, no banding
Depth	Depth32Float	Reversed-Z wants float depth (ch. 4)

Compression

Uncompressed RGBA8 costs 4 bytes per texel; a 4096^2 texture with mips is 85 MB. GPUs sample block-compressed textures directly, with no decompression pass: **ASTC** (all Apple silicon; block sizes from 4×4 at 8 bits per texel to 12×12 at 0.89) and **BC** formats (BC1–BC7, supported on Apple silicon Macs and used by cross-platform content). Use `supportsBCTextureCompression` to check, and prefer ASTC for assets you control. ASTC 4×4 is $4 \times$ smaller than RGBA8; 6×6 is $7 \times$ smaller.

6.6 Creating and binding a texture in Metal 4

Metal 4 The listing builds an sRGB texture with a full mip chain, fills level 0, lets the GPU generate the rest, creates a trilinear anisotropic sampler, and binds both through the argument table. Two Metal 4 specifics: a sampler that will be used through an argument table must be created with `supportArgumentBuffers`, and the texture must be made resident (chapter 9) before any command buffer that reads it executes.

Texture + sampler creation and binding

✓ type-checked against Apple's real metal-cpp headers · never run on a GPU

```
#include <algorithm>
#include <cmath>
#include <vector>

// Create an sRGB texture with a full mip chain from tightly packed RGBA8 pixels.
MTL::Texture* makeTexture(Renderer& r, MTL4::ComputeCommandEncoder* ce, const uint8_t* rgba,
                          uint32_t w, uint32_t h) {
    const uint32_t mips = 1 + uint32_t(std::floor(std::log2(double(std::max(w, h)))));
    auto* td = MTL::TextureDescriptor::texture2DDescriptor(MTL::PixelFormatRGBA8Unorm_sRGB, w, h, true);
    td->setMipmapLevelCount(mips);
    td->setUsage(MTL::TextureUsageShaderRead);
    td->setStorageMode(MTL::StorageModeShared); // unified memory: the CPU can write it
    MTL::Texture* tex = r.device->newTexture(td);
    td->release();

    tex->replaceRegion(MTL::Region::Make2D(0, 0, w, h), 0, rgba, 4 * w); // mip 0 only
    r.residency->addAllocation(tex);
    r.residency->commit();
    ce->generateMipmaps(tex); // GPU builds mips 1..N (in linear light)
    return tex;
}

MTL::SamplerState* makeSampler(Renderer& r) {
    auto* sd = MTL::SamplerDescriptor::alloc()->init();
    sd->setMinFilter(MTL::SamplerMinMagFilterLinear);
    sd->setMagFilter(MTL::SamplerMinMagFilterLinear);
    sd->setMipFilter(MTL::SamplerMipFilterLinear); // linear mip filter = trilinear
    sd->setMaxAnisotropy(16);
    sd->setSAddressMode(MTL::SamplerAddressModeRepeat);
    sd->setTAddressMode(MTL::SamplerAddressModeRepeat);
    sd->setSupportArgumentBuffers(true); // needed to bind by ResourceID
    MTL::SamplerState* s = r.device->newSamplerState(sd);
    sd->release();
    return s;
}
```

```
// Binding: textures and samplers go into the argument table by ResourceID.
void bindMaterial(Renderer& r, MTL::Texture* albedo, MTL::SamplerState* sampler) {
    r.args->setTexture(albedo->gpuResourceID(), 0); // [[texture(0)]]
    r.args->setSamplerState(sampler->gpuResourceID(), 0); // [[sampler(0)]]
}
```

6.7 Sampling in a shader

MSL In MSL you declare a texture with its access and sample it through a sampler object. This fragment shader also does tangent-space normal mapping. Read the comments: the normal-map decode ($2x - 1$) and the TBN construction are the two things people get wrong first.

Fragment shader: albedo sampling and tangent-space normal mapping

◇ MSL • not compiled here (needs Apple's Metal compiler)

```
#include <metal_stdlib>
using namespace metal;

struct SurfaceIn {
    float4 position [[position]];
    float3 worldNormal;
    float4 worldTangent; // xyz = tangent, w = +/-1 handedness
    float2 uv;
};

fragment float4 fs_textured(SurfaceIn in [[stage_in]],
                           texture2d<float> albedoTex [[texture(0)], // sRGB format: decoded to linear
                           texture2d<float> normalTex [[texture(1)], // plain Unorm format: raw data
                           sampler smp [[sampler(0)]] {
    float3 albedo = albedoTex.sample(smp, in.uv).rgb; // already linear here

    // Tangent-space normal map: stored as [0,1], decoded to [-1,1].
    float3 tn = normalTex.sample(smp, in.uv).xyz * 2.0 - 1.0;
    float3 N = normalize(in.worldNormal);
    float3 T = normalize(in.worldTangent.xyz - N * dot(N, in.worldTangent.xyz)); // re-orthogonalize
    float3 B = cross(N, T) * in.worldTangent.w;
    float3 worldN = normalize(T * tn.x + B * tn.y + N * tn.z); // tangent -> world

    float ndl = saturate(dot(worldN, normalize(float3(0.4, 0.8, 0.5))));
    return float4(albedo * (0.1 + 0.9 * ndl), 1.0);
}
```

Pitfall: texture looks wrong • Check in this order: (1) sRGB flag on albedo, not on normals; (2) *v* flipped (image origin is top-left here); (3) mips never generated, so distant areas shimmer; (4) normal map green channel convention (OpenGL-style versus DirectX-style assets differ); (5) sampling after divergent control flow.

Sampler cheat sheet

State	Choose
Min / mag filter	Linear for color; Nearest for IDs and pixel art
Mip filter	Linear (trilinear); NotMipmapped for UI and render targets
Address mode	Repeat for tiling, ClampToEdge for atlases and screen-space lookups, MirrorRepeat to hide seams
Max anisotropy	1 (off) to 16
Compare function	Set for shadow-map samplers (hardware PCF)

Chapter 6 in one minute

- UV is resolution-independent; texel values live at texel *centers*. Bilinear filtering is three lerps.
- Minified textures alias, so precompute mipmaps (+33% memory); the GPU picks the level from $\lambda = \log_2 \rho$ using quad derivatives, which requires uniform control flow.
- Trilinear removes level seams; anisotropic samples along the footprint's long axis for sharp grazing surfaces.
- Filter in linear space: sRGB formats for color, plain formats for data and normals. Compress with ASTC or BC.

Light and Color

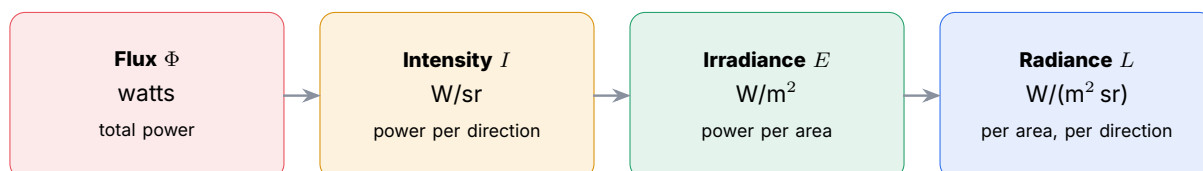
What a pixel value actually means: how light is measured, why a cosine appears in every shading equation, why 128 is not half as bright as 255, and how HDR values become something a display can show.

07

7.1 Measuring light

Rendering is the simulation of light transport, so you need a few physical quantities. Each answers a different question about the same photons.

THE FOUR QUANTITIES, FROM TOTAL POWER DOWN TO WHAT ONE PIXEL SEES



Radiance is the one that matters for rendering: it *does not change along a ray in empty space*. A pixel's value is the radiance arriving along the camera ray, which is why a ray tracer can pass radiance from surface to surface without bookkeeping for distance.

You rarely use the units. What you need is the *structure*: the shading integral at a surface point adds up incoming radiance from every direction, each weighted by the BRDF and a cosine (next section):

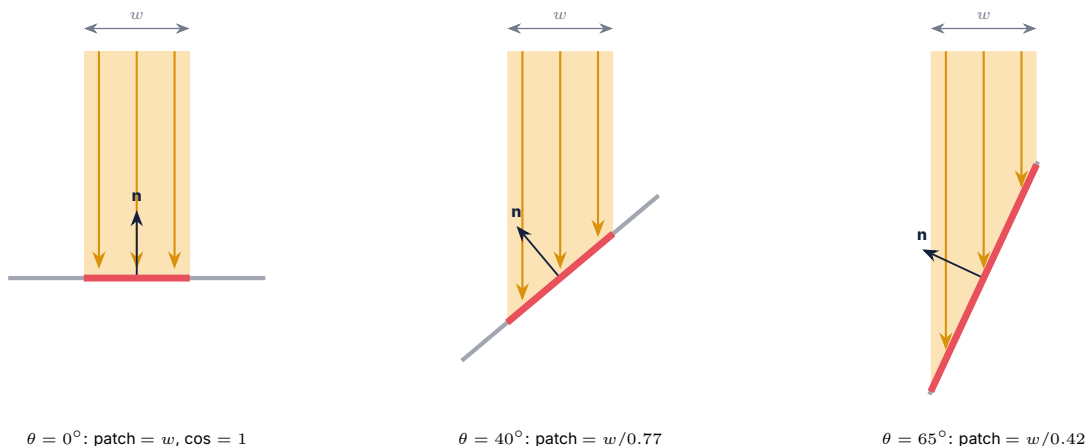
$$L_o(\mathbf{v}) = \int_{\Omega} f_r(\mathbf{l}, \mathbf{v}) L_i(\mathbf{l}) \max(0, \mathbf{n} \cdot \mathbf{l}) d\mathbf{l}$$

A real-time renderer approximates this integral with a few analytic lights plus image-based or ray-traced indirect light. Chapter 12 evaluates it with Monte Carlo.

7.2 Why there is a cosine: Lambert's law

A beam of light with fixed cross-section hitting a surface at an angle θ is spread over a larger area, $1/\cos\theta$ times as large. The same power over more area is dimmer. That is the whole reason for $\mathbf{n} \cdot \mathbf{l}$ in every lighting equation.

LAMBERT'S COSINE LAW: THE SAME BEAM LANDS ON A LONGER PATCH OF A TILTED SURFACE



The same beam (width w , so the same power) covers a patch of length $w / \cos \theta$ (red), so each bit of surface gets $\cos \theta$ times as much. Past 90° the surface faces away and receives nothing, hence $\max(0, \cdot)$.

7.3 Light falls off with distance

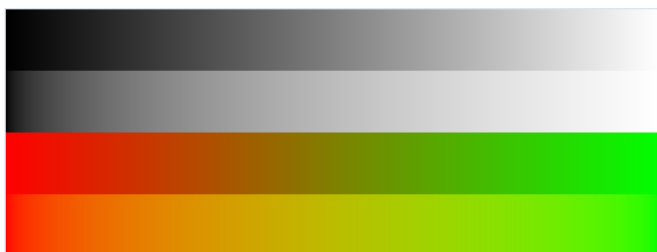
A point light's power spreads over the surface of a growing sphere, area $4\pi r^2$, so the irradiance at distance r is proportional to $1/r^2$: double the distance, a quarter of the light. Real engines usually clamp or window the curve so lights have finite reach (a cheap light needs a bounded radius), but the physical default is $1/r^2$ and it is a good one.

7.4 Gamma, sRGB and linear light

Here is the idea that causes more wrong-looking images than any other. Pixel values in an image file are **not** proportional to light. Human vision is much more sensitive to differences in dark tones than bright ones, so image formats spend more of their 256 codes on the darks. The curve used is **sRGB**, close to $\text{code} = \text{light}^{1/2.2}$. Lighting math (adding lights, multiplying by albedo, averaging samples) is only correct on **linear** values, proportional to the number of photons. So the rule is:

decode to linear → **do all the math** → **encode for the display**

SAME DATA, TWO ENCODINGS. TOP: LINEAR VALUES SHOWN RAW (TOO DARK). BELOW: ENCODED. THEN: BLENDING RED→GREEN WRONG AND RIGHT

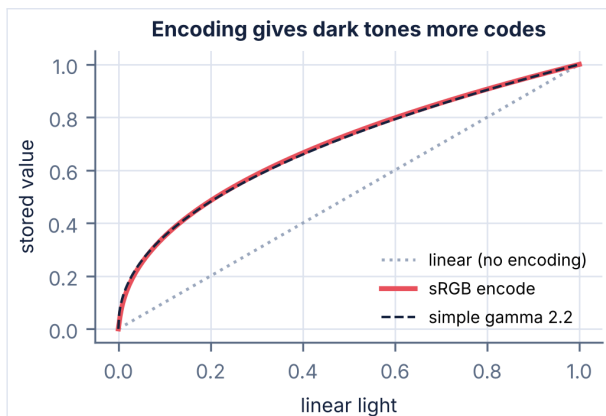


rows 1–2: a linear ramp raw versus sRGB-encoded; rows 3–4: a red→green blend of encoded values (muddy dark middle) versus blend in linear light

Row 1 is a ramp written straight into an 8-bit image: mid-gray is far darker than half. Row 2 applies the sRGB curve, which *is* perceptually even.

Rows 3 and 4: blending in encoded space dips into a dark brown halfway; blending light correctly stays bright. The same error darkens every mip level, every antialiased edge and every transparent blend.

THE sRGB CURVE: DARK TONES GET MORE OF THE 256 CODES



An 8-bit file has only 256 levels. Without the curve, the darks would band visibly. The curve is a short linear toe plus a 2.4 power; gamma 2.2" is a close approximation but not exact.

Why averaging in the wrong space is too dark

✓ compiled & run · g++13, -std=c++23 -Wall -Wextra -pedantic

```
#include <cmath>
#include <cstdio>
#include <initializer_list>

float srgbToLinear(float c) {
    return c <= 0.04045f ? c / 12.92f : std::pow((c + 0.055f) / 1.055f, 2.4f);
}
float linearToSrgb(float c) {
    return c <= 0.0031308f ? 12.92f * c : 1.055f * std::pow(c, 1.0f / 2.4f) - 0.055f;
}

int main() {
    // Mip-mapping averages texels. Average a black and a white pixel: which answer is right?
    float black = 0.0f, white = 1.0f; // as stored (sRGB-encoded) values
    float wrong = (black + white) / 2; // average the encoded numbers
    float right = linearToSrgb((srgbToLinear(black) + srgbToLinear(white)) / 2); // average the light
    std::printf("average in sRGB space : %.3f (looks too dark)\n", wrong);
    std::printf("average in linear space : %.3f (matches what your eye sees from far away)\n", right);

    std::printf("\nsRGB code -> linear light\n");
    for (int code : {0, 64, 128, 188, 255})
        std::printf(" %3d/255 = %.3f -> %.4f\n", code, code / 255.0f, srgbToLinear(code / 255.0f));
}
```

OUTPUT

```
average in sRGB space : 0.500 (looks too dark)
average in linear space : 0.735 (matches what your eye sees from far away)

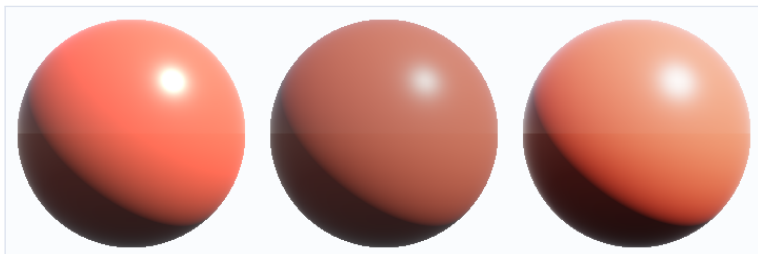
sRGB code -> linear light
 0/255 = 0.000 -> 0.0000
64/255 = 0.251 -> 0.0513
128/255 = 0.502 -> 0.2159
188/255 = 0.737 -> 0.5029
255/255 = 1.000 -> 1.0000
```

Metal does most of this for you • Choose `_sRGB` pixel formats for color textures (decoded when sampled, before filtering) and for 8-bit render targets (encoded on write, after blending). Never apply the curve by hand *and* use an sRGB target: that encodes twice and washes the image out. Your HDR intermediate targets should be linear `RGBA16Float`.

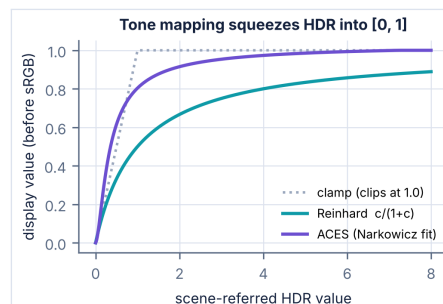
7.5 HDR, exposure and tone mapping

Real scenes span huge ranges: the sun is about 10^5 times brighter than a shaded wall. Rendering in linear floating point lets values go far above 1.0; a display can show only $[0, 1]$. **Tone mapping** compresses scene-referred values into display range. Without it, bright regions simply clip to flat white and lose all detail.

SAME SCENE, VERY BRIGHT LIGHT: CLAMPED, REINHARD, ACES



left to right: hard clamp (highlight clips), Reinhard $c/(1+c)$, ACES fit



Two common operators appear in the listing. **Reinhard** is the simplest rolling curve. The **ACES fit** (Narkowicz) adds a filmic S-shape with a toe and a shoulder, giving richer contrast and a pleasant desaturation of very bright colors. Exposure is just a multiplier before the curve: it is how you set the camera".

Tone mapping and sRGB encoding in MSL

✓ MSL - identical to code compiled & run on the CPU through a shim

```
inline float3 tonemapReinhard(float3 c) { return c / (c + 1.0f); }
inline float3 tonemapACES(float3 x) { // Krzysztof Narkowicz's ACES fit
    const float a = 2.51f, b = 0.03f, c = 2.43f, d = 0.59f, e = 0.14f;
    return saturate((x * (a * x + b)) / (x * (c * x + d) + e));
}
inline float3 linearToSRGB(float3 c) {
    c = saturate(c);
    return float3(c.x <= 0.0031308f ? 12.92f * c.x : 1.055f * pow(c.x, 1.0f / 2.4f) - 0.055f,
                  c.y <= 0.0031308f ? 12.92f * c.y : 1.055f * pow(c.y, 1.0f / 2.4f) - 0.055f,
                  c.z <= 0.0031308f ? 12.92f * c.z : 1.055f * pow(c.z, 1.0f / 2.4f) - 0.055f);
}
```

Pitfall: tone map at the end, once • Do blending, bloom, and all lighting in linear HDR, then tone map and encode as the very last step. Tone mapping early, or twice, breaks energy conservation and makes bloom look gray.

Chapter 7 in one minute

- A pixel is a measurement of *radiance* along a ray, which is constant along the ray in a vacuum.
- The cosine in every lighting formula is Lambert's law: slanted light is spread over more area. Point lights fall off as $1/r^2$.
- Files hold gamma-encoded values; do the math in linear light, using sRGB formats so hardware converts for you.
- Render in linear HDR (RGBA16Float), expose, tone map once at the end, then encode.

Materials and PBR

From Lambert and Phong to the microfacet model every modern engine uses: GGX, Fresnel, Smith shadowing, the metalness workflow, energy conservation, and the complete shading function in MSL.

08

8.1 The BRDF: how a surface reflects

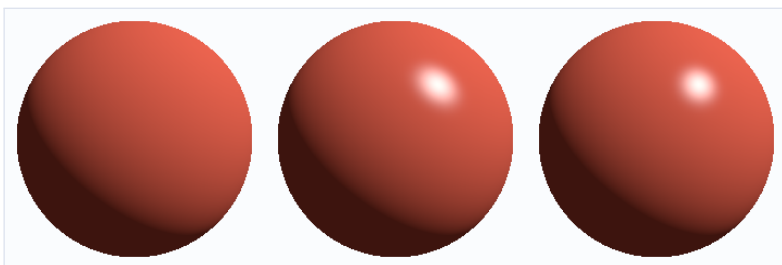
A **BRDF** (bidirectional reflectance distribution function) $f_r(\mathbf{l}, \mathbf{v})$ answers one question: if light arrives from direction $\mathbf{l}$, how much of it leaves toward the viewer in direction $\mathbf{v}$? Every material is a different BRDF. A shader's job is to evaluate it, multiply by incoming light and the cosine from chapter 7, and sum over lights.

Two physical rules constrain any BRDF that wants to look right: **reciprocity** (swap $\mathbf{l}$ and $\mathbf{v}$ and the value is the same) and **energy conservation** (a surface cannot reflect more light than arrives).

8.2 Classic models

Lambert diffuse is the constant $f_r = \text{albedo}/\pi$: light scatters equally in all directions, so the surface looks the same from any angle, and only the cosine shapes it. **Phong** and **Blinn-Phong** add a specular highlight, $(\mathbf{n} \cdot \mathbf{h})^s$ with $\mathbf{h}$ the half vector. They are cheap and artist-friendly, but their shininess exponent has no physical meaning, highlights never get the right shape at grazing angles, and they do not conserve energy.

LAMBERT ONLY, PHONG, BLINN-PHONG (RENDERED BY THE CPU SHADING CODE IN THIS BOOK)



diffuse only → a mirror-direction highlight → half-vector highlight

Good enough for 1990s games, and still fine for prototypes. Physically based rendering replaces the arbitrary exponent with a measurable *roughness* and gives the highlight the correct shape and brightness.

8.3 Microfacet theory

A real surface is rough at a scale smaller than a pixel. Model it as a field of tiny perfect mirrors (**microfacets**) with random normals. A facet reflects light from **l** to **v** only if its normal is exactly the half vector **h** = normalize(**l** + **v**). So the specular reflection is built from three terms, each answering one question:

THE COOK-TORRANCE SPECULAR BRDF: THREE QUESTIONS, THREE FUNCTIONS

$$f_{\text{spec}} = \frac{D(\mathbf{h}) F(\mathbf{v} \cdot \mathbf{h}) G(\mathbf{l}, \mathbf{v})}{4 (\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v})}$$

D: distribution

How many facets point along **h**?
Rough surfaces have a wide, dim highlight; smooth ones a tight, bright one. We use **GGX**.

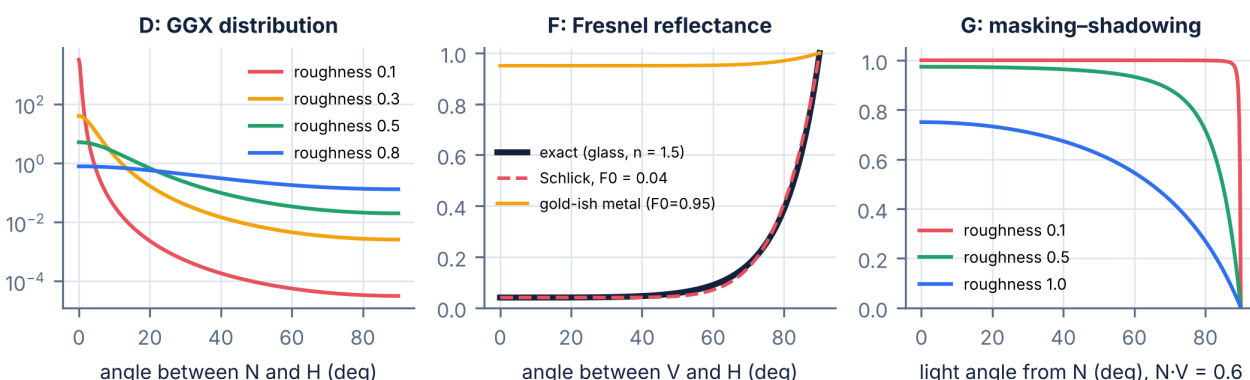
F: Fresnel

How much of the light reflects at this angle? Every surface becomes a mirror at grazing angles. We use **Schlick**.

G: shadowing

Do facets block each other from the light or the eye? Matters at rough and grazing angles. We use **Smith**.

THE ROUGHNESS PARAMETER: SHARP MIRROR TO WIDE LOBE (D, F AND G CURVES PLOTTED FROM THE LISTING'S FORMULAS)



D: GGX normal distribution

With $\alpha = \text{roughness}^2$ (the squaring spreads the artist-visible slider more evenly):

$$D_{\text{GGX}}(\mathbf{h}) = \frac{\alpha^2}{\pi ((\mathbf{n} \cdot \mathbf{h})^2 (\alpha^2 - 1) + 1)^2}$$

GGX has a sharp core and a long tail, which matches real highlights, with a soft glow around the hot spot, better than a Gaussian.

F: Fresnel and Schlick's approximation

Reflectance at normal incidence is F_0 : about 0.04 for plastic, glass, skin and most dielectrics, and 0.5–1.0 (and colored) for metals. As the angle to the surface gets shallow, reflectance rises toward 1 for everything:

$$F(\mathbf{v} \cdot \mathbf{h}) \approx F_0 + (1 - F_0) (1 - \mathbf{v} \cdot \mathbf{h})^5$$

G: Smith masking-shadowing

The visibility form $V = G / (4(\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v}))$ folds the denominator into the term, which is what the code computes (*height-correlated* Smith, because the two shadowing events are not independent).

D, V and F in MSL

✓ MSL · identical to code compiled & run on the CPU through a shim

```
// GGX / Trowbridge-Reitz normal distribution: how many microfacets face the half vector.
inline float D_GGX(float NoH, float alpha) {
    float a2 = alpha * alpha;
    float d = NoH * NoH * (a2 - 1.0f) + 1.0f;
    return a2 / (PI * d * d);
}

// Smith height-correlated visibility term  $V = G / (4 \text{ NoL NoV})$ .
inline float V_SmithGGX(float NoV, float NoL, float alpha) {
    float a2 = alpha * alpha;
    float gv = NoL * sqrt(NoV * NoV * (1.0f - a2) + a2);
    float gl = NoV * sqrt(NoL * NoL * (1.0f - a2) + a2);
    return 0.5f / max(gv + gl, 1e-5f);
}

// Schlick's Fresnel approximation.
inline float3 F_Schlick(float VoH, float3 F0) {
    return F0 + (float3(1.0f) - F0) * pow(1.0f - VoH, 5.0f);
}
```

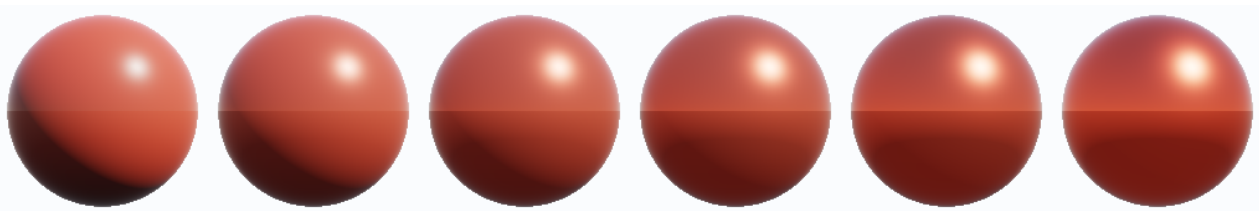
8.4 The metalness workflow

Two parameters describe nearly everything: **roughness** and **metallic**. The base color means different things depending on metallic:

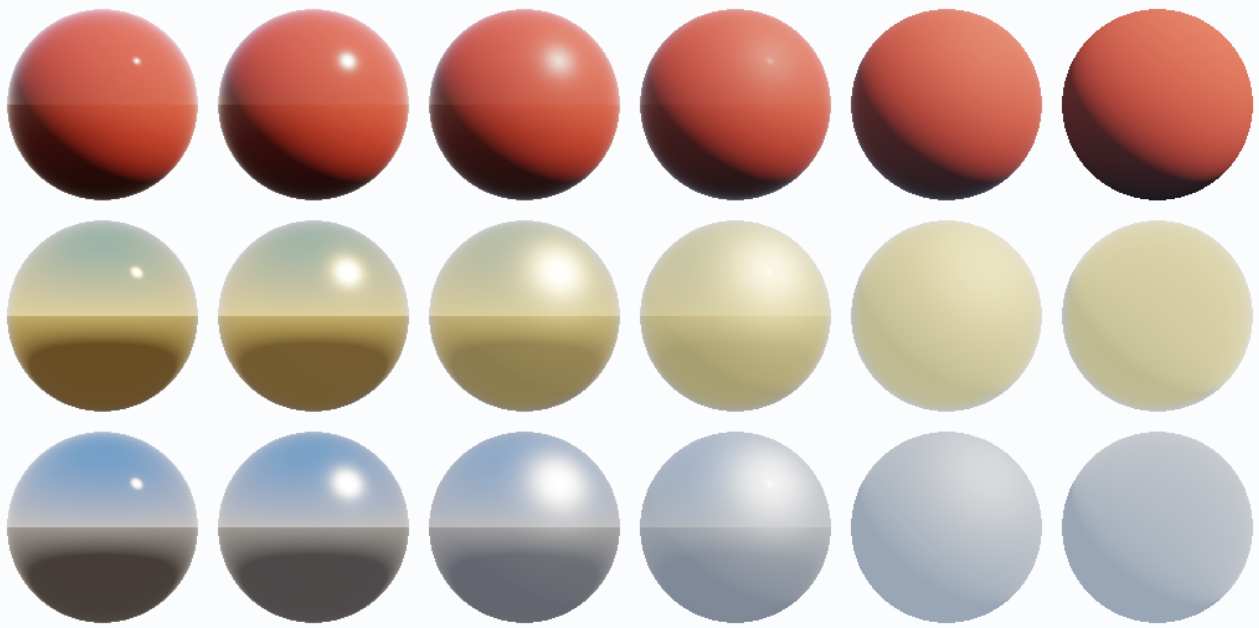
- **Dielectrics** (metallic = 0): the base color is the *diffuse* albedo. Specular is a colorless 4% with Fresnel.
- **Metals** (metallic = 1): no diffuse at all; the base color becomes F_0 , the *tint of the reflection*. Gold is yellow because its reflectance is.

So in code, $F_0 = \text{mix}(0.04, \text{baseColor}, \text{metallic})$ and the diffuse color is $\text{baseColor} \cdot (1 - \text{metallic})$.

METALLIC FROM 0 TO 1 AT FIXED ROUGHNESS 0.35: DIFFUSE FADES, THE REFLECTION TAKES THE COLOR



ROUGHNESS 0.05 TO 1.0 (LEFT TO RIGHT) FOR THREE MATERIALS: PLASTIC, GOLD, STEEL



8.5 The complete shading function

MSL Everything above fits in one short function. Note the factor $(1 - F)$ on the diffuse term: whatever energy the specular reflects, the diffuse cannot also have, a cheap approximation of energy conservation.

Complete direct-lighting evaluation for one light

✓ MSL - identical to code compiled & run on the CPU through a shim

```
// Evaluate (BRDF * cos) for one light. N, V, L are unit vectors; radiance is the light's.
inline float3 shadePBR(float3 N, float3 V, float3 L, float3 radiance, Material m) {
    float3 H = normalize(V + L);
    float NoL = saturate(dot(N, L));
    float NoV = max(dot(N, V), 1e-4f);
    float NoH = saturate(dot(N, H));
    float VoH = saturate(dot(V, H));
    float alpha = max(m.roughness * m.roughness, 0.002f);

    float3 F0 = mix(float3(0.04f), m.baseColor, m.metallic); // 4% reflectance for dielectrics
    float3 diffuse = m.baseColor * (1.0f - m.metallic) / PI; // Lambert
    float3 F = F_Schlick(VoH, F0);
    float3 spec = F * (D_GGX(NoH, alpha) * V_SmithGGX(NoV, NoL, alpha));
    float3 kd = (float3(1.0f) - F); // the rest goes to diffuse
    return (kd * diffuse + spec) * radiance * NoL;
}
```

8.6 Is it energy conserving? The furnace test

A good habit: check your BRDF numerically. Put the surface inside a uniformly lit furnace" and compute what fraction of light it reflects. It must never exceed 1. The program importance-samples the specular lobe with Fresnel set to 1 (the worst case):

White-furnace test of the GGX specular lobe

✓ compiled & run - g++ 13, -std=c++23 -Wall -Wextra -pedantic

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <numbers>
#include <random>
```

```
constexpr float pi = std::numbers::pi_v<float>;

float D_GGX(float NoH, float a) {
    float a2 = a * a, d = NoH * NoH * (a2 - 1) + 1;
    return a2 / (pi * d * d);
}

float V_Smith(float NoV, float NoL, float a) { // height-correlated visibility = G / (4 NoL NoV)
    float a2 = a * a;
    float gv = NoL * std::sqrt(NoV * NoV * (1 - a2) + a2);
    float gl = NoV * std::sqrt(NoL * NoL * (1 - a2) + a2);
    return 0.5f / (gv + gl);
}

// Directional albedo of the specular lobe with Fresnel F = 1: the fraction of light it reflects.
// A physically plausible BRDF must return <= 1. Importance-sample the GGX half vector.
float albedo(float NoV, float roughness, int N, std::mt19937& rng) {
    std::uniform_real_distribution<float> U(0.0f, 1.0f);
    float a = roughness * roughness, sum = 0;
    float vx = std::sqrt(1 - NoV * NoV), vz = NoV; // view vector in the normal's frame (y = 0)
    for (int i = 0; i < N; ++i) {
        float u1 = U(rng), u2 = U(rng);
        float cosH = std::sqrt((1 - u2) / (1 + (a * a - 1) * u2));
        float sinH = std::sqrt(std::max(0.0f, 1 - cosH * cosH)), phi = 2 * pi * u1;
        float hx = sinH * std::cos(phi), hy = sinH * std::sin(phi), hz = cosH;
        float VoH = vx * hx + vz * hz;
        float lx = 2 * VoH * hx - vx, ly = 2 * VoH * hy,
            lz = 2 * VoH * hz - vz; // L = reflect V about H
        (void)ly;
        if (lz <= 0 || VoH <= 0) continue; // reflected below the surface: lost
        // weight = f * cos / pdf, with pdf(L) = D * NoH / (4 * VoH)
        sum += 4 * V_Smith(NoV, lz, a) * lz * VoH / hz;
        (void)lx;
    }
    return sum / N;
}

int main() {
    std::mt19937 rng(1);
    std::printf("roughness : albedo at NoV = 0.9, 0.5, 0.1      (1.00 = no energy lost)\n");
    for (float r : {0.1f, 0.3f, 0.5f, 0.7f, 1.0f})
        std::printf("  %.1f      :   %.3f   %.3f   %.3f\n", r, albedo(0.9f, r, 200000, rng),
            albedo(0.5f, r, 200000, rng), albedo(0.1f, r, 200000, rng));
}
```

OUTPUT

```
roughness : albedo at NoV = 0.9, 0.5, 0.1      (1.00 = no energy lost)
0.1      :   1.000   1.000   0.995
0.3      :   0.990   0.974   0.887
0.5      :   0.908   0.858   0.899
0.7      :   0.692   0.705   0.860
1.0      :   0.330   0.451   0.758
```

Two honest observations. First, no value exceeds 1.0, so the model does not invent light. Second, *rough* lobes return as little as 33% at roughness = 1: single-scattering microfacet models lose the light that bounces between facets. Production engines add a multiple-scattering compensation (a small energy-gain factor derived from a lookup table); it is an upgrade for chapter 15's checklist, not needed on day one.

Where the numbers come from • Reflectance values (0.04 for dielectrics, gold's F_0) are standard measured figures; the 0.002 minimum on α in the code avoids dividing by zero for perfect mirrors. Both are choices you can change, the structure is what matters.

Material authoring cheat sheet

Material	metallic / roughness	Notes
Plastic, painted wall	0 / 0.3–0.8	Base color is albedo (not pure black or white: use 0.02–0.9)
Rubber, cloth	0 / 0.8–1	Cloth often needs a dedicated sheen BRDF
Polished metal	1 / 0.1–0.3	Base color is the reflection tint
Rusty / dirty metal	mask $0 \leftrightarrow 1$	Metalness is almost binary; blend at the edges only
Glass, water	0 / 0.0–0.1	Needs transmission; out of scope here

Chapter 8 in one minute

- A BRDF gives reflected light for each light/view pair; it must be reciprocal and conserve energy.
- The microfacet model is $\frac{DFG}{4N \cdot L N \cdot V}$: GGX distribution, Schlick Fresnel, Smith shadowing.
- Roughness $\alpha = r^2$. Metalness picks diffuse versus colored reflection: $F_0 = \text{mix}(0.04, \text{base}, m)$.
- Verify with a furnace test; single-scatter GGX loses energy at high roughness.

Metal 4 Fundamentals

The object model, the frame loop, and the three rules that replace Metal's old automatic bookkeeping: you keep memory resident, you synchronize with barriers, and you reuse command allocators only after the GPU is done.



9.1 What changed in Metal 4

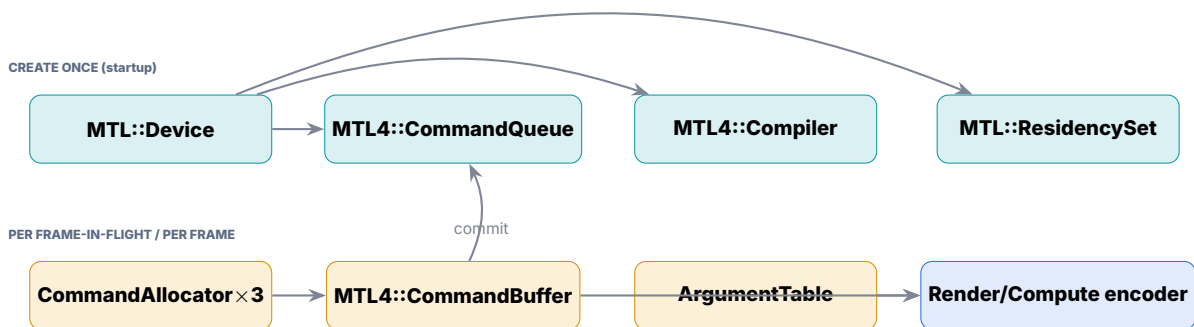
Earlier Metal tracked resources, lifetimes and hazards for you. Metal 4 hands that control to you in exchange for lower CPU overhead and a design that suits engines with a render graph. The practical differences you meet first:

Old Metal versus Metal 4 (the pieces this guide uses)		
Concern	Earlier Metal	Metal 4
Command memory	Hidden inside command buffers	MTL4::CommandAllocator, reset by you once the GPU is done
Command buffers	Created per frame from the queue	Created from the device, reused; beginCommandBuffer
Binding resources	setBytes, setTexture per encoder	MTL4::ArgumentTable and GPU addresses
Resource lifetime	Tracked by command buffers	MTL::ResidencySet you maintain
Hazards between passes	Tracked automatically	Explicit barriers
Pipeline compile	device->newRenderPipelineState	MTL4::Compiler

About Metal 4.1 and macOS 27 • Apple's newest releases (Metal 4.1, MSL 4.1) reportedly add features beyond what this guide's code needs. I have not verified their specifics against Apple's documentation or headers, so this guide sticks to the Metal 4 API that I type-checked against the real metal-cpp headers. Check Apple's release notes for anything newer.

9.2 The object model at a glance

WHO OWNS WHAT: THE OBJECTS YOU CREATE ONCE, AND THE ONES THAT CHANGE EVERY FRAME



The device makes everything. The **allocator** is the memory the commands are written into: you need one per frame in flight, because the GPU may still be reading frame N 's commands while you record $N+1$. The **argument table** is where you write the addresses and textures a draw will use.

Create the device, queue, allocators, compiler, argument table and residency set

✓ type-checked against Apple's real metal-cpp headers · never run on a GPU

```

// Compile as C++23; link -framework Metal -framework Foundation -framework QuartzCore.
// In exactly ONE .cpp file, define the *_PRIVATE_IMPLEMENTATION macros before the includes:
#define NS_PRIVATE_IMPLEMENTATION
#define MTL_PRIVATE_IMPLEMENTATION
#define CA_PRIVATE_IMPLEMENTATION
#include <Foundation/Foundation.hpp>
#include <Metal/Metal.hpp>
#include <QuartzCore/QuartzCore.hpp>

constexpr int kFramesInFlight = 3;

struct Renderer {
    MTL::Device* device = nullptr;
    MTL4::CommandQueue* queue = nullptr;
    MTL4::CommandBuffer* cmd = nullptr; // one reusable command buffer
    MTL4::CommandAllocator* alloc[kFramesInFlight] = {}; // one allocator per frame in flight
    MTL4::Compiler* compiler = nullptr;
    MTL4::ArgumentTable* args = nullptr; // where shader bindings live
    MTL::ResidencySet* residency = nullptr; // everything the GPU may touch
    MTL::SharedEvent* frameEvent = nullptr; // signaled when a frame's GPU work ends
    CA::MetalLayer* layer = nullptr; // from your NSView (AppKit code not shown)
    uint64_t frame = 0;
};

bool initRenderer(Renderer& r, CA::MetalLayer* layer) {
    NSError* err = nullptr;
    r.device = MTL::CreateSystemDefaultDevice();
    r.queue = r.device->newMTL4CommandQueue();
    r.cmd = r.device->newCommandBuffer();
    for (auto& a : r.alloc) a = r.device->newCommandAllocator();

    auto* cd = MTL4::CompilerDescriptor::alloc()->init();
    r.compiler = r.device->newCompiler(cd, &err);
    cd->release();

    auto* ad = MTL4::ArgumentTableDescriptor::alloc()->init();
    ad->setMaxBufferBindCount(16); // buffer slots 0..15
    ad->setMaxTextureBindCount(16);
    r.args = r.device->newArgumentTable(ad, &err);
    ad->release();

    auto* rd = MTL::ResidencySetDescriptor::alloc()->init();
    r.residency = r.device->newResidencySet(rd, &err);
    rd->release();
    r.queue->addResidencySet(r.residency); // the queue keeps these resources resident
    r.queue->addResidencySet(layer->residencySet()); // ...and so does the swapchain's set
  
```

```

r.frameEvent = r.device->newSharedEvent();
r.layer = layer;
layer->setDevice(r.device);
layer->setPixelFormat(MTL::PixelFormatBGRA8Unorm_sRGB);
return r.queue && r.cmd && r.compiler && r.args && r.residency && err == nullptr;
}

```

All Metal objects in metal-cpp are manually reference counted: anything you `alloc()->init()` or get from a `new...` function you own and must `release()`. Descriptors are short-lived, so release them right after creating the object that consumed them, as the listing does. (Chapter 5 of the C++ guide shows how to wrap this in a RAII smart pointer.)

9.3 Residency: the GPU may only touch memory you declared

Metal 4 does not keep resources alive or mapped just because a command mentions them. You put every buffer and texture in a **residency set**, call `commit()`, and attach the set to the queue. A resource that is not resident is a GPU fault, usually shown as garbage or a hang rather than a clean error.

THE RESIDENCY WORKFLOW

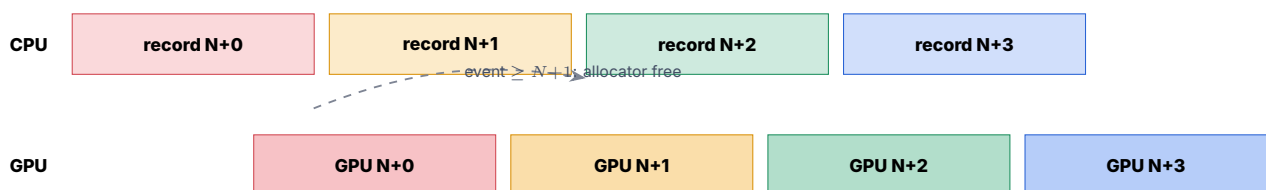


Batch your additions (a whole level load, then one `commit()`). Remove allocations before releasing the resource. The drawable's own set (`layer->residencySet()`) is added separately, as in the setup listing.

9.4 The frame loop and CPU–GPU synchronization

The CPU runs ahead of the GPU. Letting it run ahead by up to three frames hides jitter; going further adds latency. The bookkeeping that makes this safe is one shared event counter:

THREE FRAMES IN FLIGHT: THE CPU RECORDS FRAME $N+2$ WHILE THE GPU STILL DRAWS FRAME N



Before reusing allocator slot s , the CPU waits until the shared event reaches the value signaled when the frame that last used s finished. With three slots, the CPU only stalls if the GPU falls three frames behind.

One frame: wait, reset, record, submit, signal, present

✓ type-checked against Apple's real metal-cpp headers · never run on a GPU

```

void encodeScene(Renderer& r, MTL::RenderCommandEncoder* enc); // next section

void renderFrame(Renderer& r) {
    const uint64_t slot = r.frame % kFramesInFlight;

    // 1. The allocator we are about to reuse must no longer be in use by the GPU.
    if (r.frame >= kFramesInFlight)
        r.frameEvent->waitUntilSignaledValue(r.frame - kFramesInFlight + 1, 1000 /*ms*/);
    r.alloc[slot]->reset();

    CA::MetalDrawable* drawable = r.layer->nextDrawable();
}

```

```

    if (!drawable) return;

    // 2. Record. The same MTL4::CommandBuffer object is re-begun every frame.
    r.cmd->beginCommandBuffer(r.alloc[slot]);

    auto* pass = MTL4::RenderPassDescriptor::alloc()->init();
    auto* color = pass->colorAttachments()->object(0);
    color->setTexture(drawable->texture());
    color->setLoadAction(MTL::LoadActionClear);
    color->setStoreAction(MTL::StoreActionStore);
    color->setClearColor(MTL::ClearColor::Make(0.04, 0.06, 0.11, 1.0));

    MTL4::RenderCommandEncoder* enc = r.cmd->renderCommandEncoder(pass);
    encodeScene(r, enc);
    enc->endEncoding();
    r.cmd->endCommandBuffer();
    pass->release();

    // 3. Submit. The drawable's texture is only safe to write once the display is done with it.
    r.queue->wait(drawable);
    const MTL4::CommandBuffer* buffers[] = { r.cmd };
    r.queue->commit(buffers, 1);
    r.queue->signalDrawable(drawable);
    drawable->present();
    r.queue->signalEvent(r.frameEvent, r.frame + 1);    // "frame N is finished" for the CPU
    ++r.frame;
}

```

Read the order of the last four lines carefully • **(1)** The queue waits on the drawable before the commit, so you never write a texture the display still uses. **(2)** Commit. **(3)** Signal the drawable so presentation can wait for your rendering. **(4)** Signal your own event with the frame number. Mixing these up is the classic cause of flicker or an indefinite hang.

9.5 Barriers: saying which work waits for which

Inside a command buffer the GPU is free to overlap passes that look sequential. If pass B reads what pass A wrote, you must say so with a **barrier**: everything in stage X of earlier work finishes before stage Y of later work starts". Metal 4 offers three scopes, from narrow to wide:

Barrier calls (all take stage masks and a visibility option)

Call	Meaning
<code>barrierAfterEncoderStages</code>	Within this encoder: later commands wait for earlier ones (e.g. dispatch → dispatch)
<code>barrierAfterQueueStages</code>	This encoder's later stages wait for earlier encoders' stages in the queue
<code>barrierAfterStages</code>	On the command buffer, between whole encoders

The stages are values like `MTL::StageDispatch`, `MTL::StageFragment`, `MTL::StageBlit` and `MTL::StageAccelerationStructure`. Make the *narrowest* barrier that is correct: a barrier that is too wide serializes work the GPU could overlap. Chapter 14's render graph computes them automatically from declared reads and writes, and chapter 11 shows two real examples.

9.6 Ask the device what it supports

Capability queries before choosing a code path

✓ type-checked against Apple's real metal-cpp headers · never run on a GPU

```
#include <stdio>

// Ask the device what it can do before choosing a code path. Never assume from the marketing name.
void printCaps(MTL::Device* d) {
    std::printf("%s\n", d->name()->utf8String());
    std::printf("  Metal 4 family      : %d\n", d->supportsFamily(MTL::GPUFamilyMetal4));
    std::printf("  Apple9+ (M3 class)   : %d\n", d->supportsFamily(MTL::GPUFamilyApple9));
    std::printf("  Apple10             : %d\n", d->supportsFamily(MTL::GPUFamilyApple10));
    std::printf("  ray tracing         : %d (from render stages: %d)\n",
        d->supportsRaytracing(), d->supportsRaytracingFromRender());
    std::printf("  unified memory      : %d\n", d->hasUnifiedMemory());
    std::printf("  working set         : %.1f GiB\n", d->recommendedMaxWorkingSetSize() / 1073741824.0);
}
```

Chapter 9 in one minute

- Metal 4 gives you three jobs: keep resources resident, record into allocators you reset only when the GPU is done, and write explicit barriers.
- A frame: wait on the event, reset the allocator, begin the buffer, encode, end, wait on the drawable, commit, signal the drawable, present, signal your event.
- Argument tables hold GPU addresses and textures; there is no `setBytes`, so keep an upload ring buffer.
- Query capabilities (family, ray tracing, BC compression) instead of guessing from the chip name.

Your First Pipeline

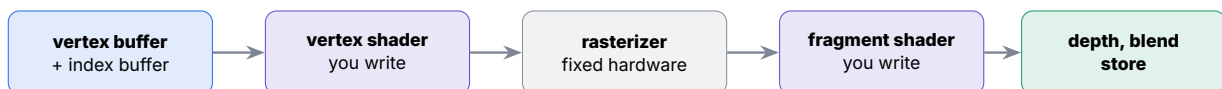
A triangle mesh on screen, step by step: vertex pulling, structs that match across C++ and MSL, depth state, upload ring buffers and the draw call. This chapter covers the whole path from data to pixel.

10

10.1 What a render pipeline is

A **render pipeline state** (PSO) bundles everything fixed for a draw: the vertex and fragment functions, the pixel formats of the targets, blending, and sample count. Creating one compiles shaders to GPU machine code, which is slow, so you build pipelines at load time and cache them, never mid-frame. In Metal 4 the compiler object does it.

FROM YOUR DATA TO A PIXEL: WHO DOES WHAT



The PSO fixes the programmable stages and target formats. The **argument table** supplies addresses. The **depth-stencil state** configures the test at the right. The draw call says how many vertices to run.

10.2 Vertex pulling: no vertex descriptor

The classic way to feed vertices uses a *vertex descriptor* that describes attribute formats. Modern engines mostly skip it: the vertex shader receives a pointer to the vertex array and the vertex ID, and *pulls* whatever it needs. This is simpler, more flexible (one buffer can hold many layouts), and the way ray-tracing shaders read geometry too.

10.3 Structs must match on both sides

The C++ `Uniforms` and the MSL `Uniforms` are two separate declarations of the same bytes. If they disagree, nothing warns you; the picture is just wrong. The most common trap is `float3`.

THE float3 TRAP: IN MSL, float3 IS 16 BYTES WIDE WITH 16-BYTE ALIGNMENT. packed_float3 IS 12.

struct { float3 position; float3 normal; } = 32 bytes



struct { packed_float3 position; packed_float3 normal; } = 24 bytes



Your C++ struct of six floats is 24 bytes. If the shader declares `float3`, it reads the wrong bytes for every vertex after the first. Use `packed_float3`, or declare four-float vectors and pad on both sides.

The same care applies to matrices: `float4x4` is column-major and 16-byte aligned. A `static_assert(sizeof(Uniforms)==80)` on the C++ side turns a silent mismatch into a compile error.

10.4 Shaders

MSL The vertex shader reads the vertex by ID, transforms with the model-view-projection matrix (matrix on the left, per chapter 3), and passes the normal on. The fragment shader lights it with a simple Lambert term. Because the target is an `_sRGB` format, the shader returns *linear* color and the hardware encodes it (chapter 7).

Vertex pulling and a Lambert fragment shader

◇ MSL · not compiled here (needs Apple's Metal compiler)

```
#include <metal_stdlib>
using namespace metal;

struct Vertex {
    packed_float3 position;
    packed_float3 normal;
}; // 24 bytes, matches the C++ struct
struct Uniforms {
    float4x4 mvp;
    float4 lightDir;
}; // 80 bytes; float4x4 is column-major

struct VSOut {
    float4 position [[position]]; // clip-space position (SV_Position)
    float3 normal;                // interpolated across the triangle
};

// "Vertex pulling": read vertices straight from a buffer by index, no vertex descriptor needed.
vertex VSOut vs_main(uint vid [[vertex_id]], device const Vertex* verts [[buffer(0)]],
                    constant Uniforms& u [[buffer(1)]]) {
    Vertex v = verts[vid]; // indexed draws: vid is the index value
    VSOut o;
    o.position = u.mvp * float4(v.position, 1.0); // matrix on the left: column vectors
    o.normal = v.normal;
    return o;
}

fragment float4 fs_main(VSOut in [[stage_in]], constant Uniforms& u [[buffer(1)]]) {
    float ndl = saturate(dot(normalize(in.normal), -u.lightDir.xyz));
    float3 color = float3(0.25, 0.55, 0.95) * (0.15 + 0.85 * ndl); // linear; the sRGB target encodes it
    return float4(color, 1.0);
}
```

10.5 Building the pipeline and depth state

Library loading, pipeline creation through the Metal 4 compiler, depth state and depth texture

✓ type-checked against Apple's real metal-cpp headers

```
MTL::Library* loadLibrary(Renderer& r, const char* path) { // a .metallib built by xcrun metal
    NSError* err = nullptr;
    return r.device->newLibrary(NS::String::string(path, NS::UTF8StringEncoding), &err);
}

static MTL4::LibraryFunctionDescriptor* fnDesc(MTL::Library* lib, const char* name) {
    auto* d = MTL4::LibraryFunctionDescriptor::alloc()->init();
    d->setLibrary(lib);
    d->setName(NS::String::string(name, NS::UTF8StringEncoding));
    return d;
}

// Pipelines are compiled by the MTL4 compiler object, not by the device.
MTL::RenderPipelineState* makePipeline(Renderer& r, MTL::Library* lib, const char* vertexName,
                                       const char* fragmentName, MTL::PixelFormat colorFormat) {
    auto* vs = fnDesc(lib, vertexName);
    auto* fs = fnDesc(lib, fragmentName);
    auto* pd = MTL4::RenderPipelineDescriptor::alloc()->init();
    pd->setVertexFunctionDescriptor(vs);
    pd->setFragmentFunctionDescriptor(fs);
    pd->colorAttachments()->object(0)->setPixelFormat(colorFormat);

    NSError* err = nullptr;
    MTL::RenderPipelineState* pso = r.compiler->newRenderPipelineState(pd, nullptr, &err);
}
```

```

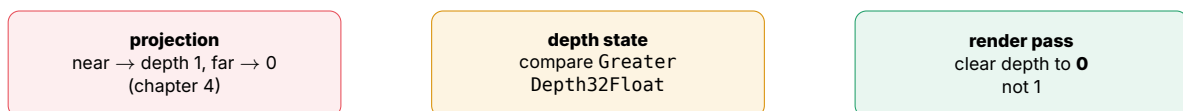
    pd->release();
    vs->release();
    fs->release();
    return pso; // nullptr on failure; read `err`
}

MTL::DepthStencilState* makeDepthState(Renderer& r, bool reverseZ) {
    auto* dd = MTL::DepthStencilDescriptor::alloc()->init();
    dd->setDepthCompareFunction(reverseZ ? MTL::CompareFunctionGreater : MTL::CompareFunctionLess);
    dd->setDepthWriteEnabled(true);
    MTL::DepthStencilState* s = r.device->newDepthStencilState(dd);
    dd->release();
    return s;
}

MTL::Texture* makeDepthTexture(Renderer& r, uint32_t w, uint32_t h) {
    auto* td = MTL::TextureDescriptor::texture2DDescriptor(MTL::PixelFormatDepth32Float, w, h, false);
    td->setUsage(MTL::TextureUsageRenderTarget);
    td->setStorageMode(MTL::StorageModePrivate);
    MTL::Texture* t = r.device->newTexture(td);
    r.residency->addAllocation(t); // GPU may only touch resident memory
    r.residency->commit();
    return t;
}

```

REVERSED-Z IN METAL: THREE SETTINGS THAT MUST AGREE



Miss one and you see nothing, or only things behind everything else. If the scene is invisible after switching to reversed-Z, check the clear value first.

10.6 Per-draw data: the upload ring

There is no `setBytes` in Metal 4, so small per-draw data lives in a buffer. The simplest scheme is a **ring**: one big shared buffer per frame in flight, a head offset that bumps for each push, and a reset when that frame slot comes around again. The shader receives the address of its slice.

An upload ring, a Mesh, and the draw call ✓ type-checked against Apple's real metal-cpp headers · never run on a GPU

```

#include <cstring> // memcpy

// A bump allocator over one shared buffer: per-draw constants live here, addressed by GPU pointer.
struct UploadRing {
    MTL::Buffer* buffer = nullptr;
    size_t head = 0;

    struct Slice { void* cpu; MTL::GPUAddress gpu; };
    Slice push(const void* data, size_t bytes) {
        head = (head + 255) & ~size_t(255); // keep 256-byte alignment
        Slice s{static_cast<char*>(buffer->contents()) + head, buffer->gpuAddress() + head};
        std::memcpy(s.cpu, data, bytes);
        head += bytes;
        return s;
    }
    void reset() { head = 0; } // call once per frame slot
};

struct Uniforms { float mvp[16]; float lightDir[4]; }; // matches the MSL struct, 80 bytes

struct Mesh { MTL::Buffer* vertices; MTL::Buffer* indices; uint32_t indexCount; };

```

```

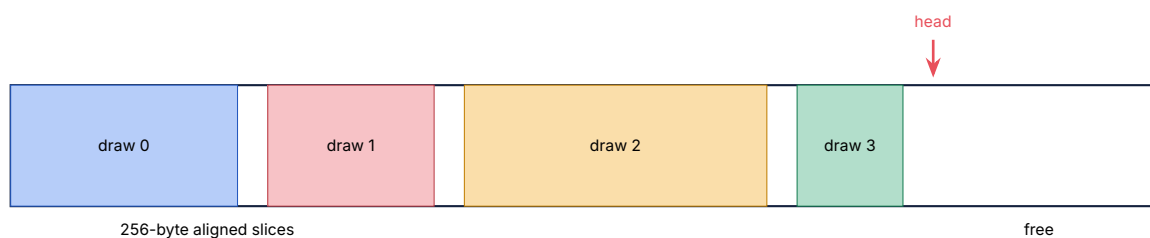
void drawMesh(Renderer& r, MTL4::RenderCommandEncoder* enc, UploadRing& ring,
              const Mesh& mesh, const Uniforms& u,
              MTL::RenderPipelineState* pso, MTL::DepthStencilState* depth) {
    enc->setRenderPipelineState(pso);
    enc->setDepthStencilState(depth);
    enc->setCullMode(MTL::CullModeBack);
    enc->setFrontFacingWinding(MTL::WindingCounterClockwise);

    auto uni = ring.push(&u, sizeof u);
    r.args->setAddress(mesh.vertices->gpuAddress(), 0);           // [[buffer(0)]] in the shader
    r.args->setAddress(uni.gpu, 1);                               // [[buffer(1)]]
    enc->setArgumentTable(r.args, MTL::RenderStageVertex | MTL::RenderStageFragment);

    enc->drawIndexedPrimitives(MTL::PrimitiveTypeTriangle, mesh.indexCount, MTL::IndexTypeUInt32,
                              mesh.indices->gpuAddress(), mesh.indexCount * sizeof(uint32_t));
}

```

THE UPLOAD RING FOR ONE FRAME SLOT (HEAD ADVANCES; RESET AT THE NEXT USE OF THE SLOT)



Pass the slice's GPU address to the shader through the argument table. Never overwrite a slot the GPU is still reading: that is why there is one ring per frame in flight.

Debugging a black screen, in order • Is the layer's pixel format set? Is the depth clear value right for your convention? Is the cull winding opposite your mesh? Is the vertex buffer resident? Does the C++ struct size equal the MSL size? Is `drawIndexedPrimitives` passed the *index* buffer's address? Capture one frame in Xcode's Metal debugger and inspect the first draw.

Chapter 10 in one minute

- A pipeline state (compiled by the Metal 4 compiler, cached at load) fixes shaders, target formats and blending; depth state is separate.
- Pull vertices from a buffer by ID. Keep C++ and MSL structs byte-identical, mind `float3` padding, and assert sizes.
- Reversed-Z needs three settings that agree: projection, Greater compare, clear to 0.
- Per-draw data lives in an upload ring addressed by GPU pointers.

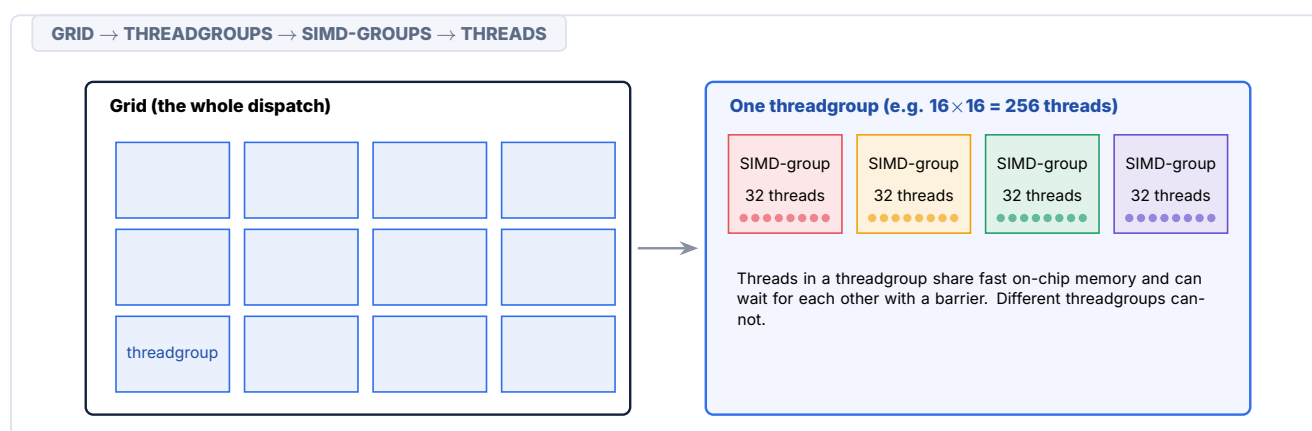
Compute and Parallelism

How a GPU runs thousands of threads: threadgroups, SIMD-groups, shared memory, barriers and a worked parallel algorithm (prefix sum). Compute is the foundation for culling, post-processing, ray tracing and the render graph.

11

11.1 The GPU execution model

A compute **dispatch** launches a grid of threads. They are organized in a hierarchy that mirrors the hardware:



- **SIMD-group** (warp): 32 threads on Apple GPUs that execute the *same instruction* in lockstep. Special `simd_*` functions let them exchange data with no memory traffic.
- **Threadgroup**: a block of threads (up to 1024) scheduled on one GPU core. They share threadgroup memory (a fast on-chip scratchpad) and can synchronize with `threadgroup_barrier`.
- **Grid**: all threadgroups of one dispatch. They run in any order, possibly at different times, so you must not rely on one threadgroup finishing before another starts.

Divergence • Because a SIMD-group executes one instruction at a time, an `if` where some lanes go one way and some the other runs *both* paths with lanes masked off. Keep neighboring threads on the same path where you can.

11.2 Pick a threadgroup size

Use a multiple of 32. A good default for 2D image work is 16×16 (256) or 8×8 (64); for 1D work, 128 or 256. Divide the problem size by the threadgroup size, *rounding up*, as in $(w + 15)/16$, and guard against threads that fall off the edge. Or use `dispatchThreads`, which handles the remainder for you.

11.3 Writing the compute pipeline in Metal 4

Metal 4 A compute pipeline is created through the same compiler object. The listing then chains two dependent dispatches (a blur is two passes) with a barrier between them, and a barrier after the compute work so a following render pass may read the result.

Compute pipeline creation and two dependent dispatches with barriers

✓ type-checked against Apple's real metal-cpp headers · never run on a GPU

```
MTL::ComputePipelineState* makeCompute(Renderer& r, MTL::Library* lib, const char* name) {
    auto* fd = MTL4::LibraryFunctionDescriptor::alloc()->init();
    fd->setLibrary(lib);
    fd->setName(NS::String::string(name, NS::UTF8StringEncoding));
    auto* cd = MTL4::ComputePipelineDescriptor::alloc()->init();
    cd->setComputeFunctionDescriptor(fd);
    NS::Error* err = nullptr;
    MTL::ComputePipelineState* pso = r.compiler->newComputePipelineState(cd, nullptr, &err);
    cd->release();
    fd->release();
    return pso;
}

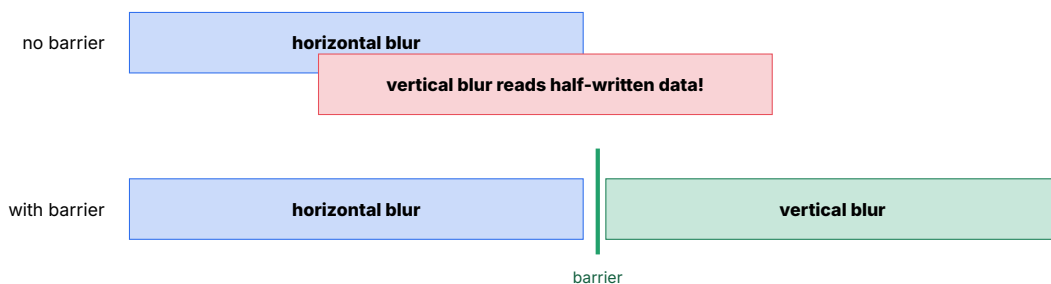
// Two dependent compute passes in ONE compute encoder, then a render pass that reads the result.
void encodeBlurThenDraw(Renderer& r, MTL::ComputePipelineState* horiz, MTL::ComputePipelineState* vert,
    MTL::Texture* tex, MTL4::RenderPassDescriptor* pass) {
    r.args->setTexture(tex->gpuResourceID(), 0); // [[texture(0)]]

    MTL4::ComputeCommandEncoder* ce = r.cmd->computeCommandEncoder();
    ce->setArgumentTable(r.args);
    const MTL::Size groups = MTL::Size::Make((tex->width() + 15) / 16, (tex->height() + 15) / 16, 1);
    const MTL::Size threads = MTL::Size::Make(16, 16, 1);

    ce->setComputePipelineState(horiz);
    ce->dispatchThreadgroups(groups, threads);
    // Nothing is tracked for you in Metal 4: say "dispatches finished before the next dispatch starts".
    ce->barrierAfterEncoderStages(MTL::StageDispatch, MTL::StageDispatch, MTL4::VisibilityOptionDevice);
    ce->setComputePipelineState(vert);
    ce->dispatchThreadgroups(groups, threads);
    ce->endEncoding();

    MTL4::RenderCommandEncoder* enc = r.cmd->renderCommandEncoder(pass);
    // "Fragment work in THIS encoder waits for earlier dispatch work in the queue."
    enc->barrierAfterQueueStages(MTL::StageDispatch, MTL::StageFragment, MTL4::VisibilityOptionDevice);
    // ... draw calls that sample `tex` ...
    enc->endEncoding();
}
```

WHY THE BARRIER IS NEEDED: TWO DISPATCHES THAT LOOK SEQUENTIAL MAY OVERLAP



11.4 A worked algorithm: parallel prefix sum

A **prefix sum** (scan) turns $[3, 1, 7, 0, 4, 1, 6, 3]$ into $[0, 3, 4, 11, 11, 15, 16, 22]$: each output is the sum of everything before it. It looks inherently sequential, yet GPUs do it in $O(\log n)$ rounds, and it is the building block for stream compaction, sorting, culling and building BVHs. Here is the classic work-efficient version (Blelloch), written on the CPU the way a threadgroup executes it so you can see each round:

BLELLOCH SCAN: AN UP-SWEEP BUILDS A TREE OF PARTIAL SUMS, A DOWN-SWEEP DISTRIBUTES THEM

input	3	1	7	0	4	1	6	3
up 1	3	4	7	7	4	5	6	9
up 2	3	4	7	11	4	5	6	14
up 3	3	4	7	11	4	5	6	25
clear root	3	4	7	11	4	5	6	0
down	3	4	7	0	4	5	6	11
down	3	0	7	4	4	11	6	16
result	0	3	4	11	11	15	16	22

Blelloch exclusive scan, each round is a threadgroup barrier

✓ compiled & run · g++ 13, -std=c++23 -Wall -Wextra -pedantic

```
#include <cstdio>
#include <numeric>
#include <vector>

// Blelloch work-efficient exclusive scan, written the way a threadgroup executes it:
// each "step" is one parallel round, and the end of a round is a threadgroup barrier.
void show(const char* label, const std::vector<int>& a) {
    std::printf("%-12s", label);
    for (int v : a) std::printf("%3d", v);
    std::puts("");
}

int main() {
    std::vector<int> a{3, 1, 7, 0, 4, 1, 6, 3};
    const int n = int(a.size());
    const std::vector<int> input = a;
    show("input", a);

    for (int stride = 1; stride < n; stride *= 2) {
        for (int t = 0; t < n / (2 * stride); ++t) {
            int i = (t + 1) * 2 * stride - 1;
            a[i] += a[i - stride];
        }
        show("up-sweep", a);
    }
    a[n - 1] = 0;
    for (int stride = n / 2; stride >= 1; stride /= 2) {
        for (int t = 0; t < n / (2 * stride); ++t) {
            int i = (t + 1) * 2 * stride - 1;
            int left = a[i - stride];
            a[i - stride] = a[i];
            a[i] += left;
        }
        show("down-sweep", a);
    }
    std::vector<int> ref(n);
    std::exclusive_scan(input.begin(), input.end(), ref.begin(), 0);
    show("reference", ref);
    std::puts(a == ref ? "match" : "MISMATCH");
}
```

OUTPUT

```

input      3  1  7  0  4  1  6  3
up-sweep   3  4  7  7  4  5  6  9
up-sweep   3  4  7 11  4  5  6 14
up-sweep   3  4  7 11  4  5  6 25
down-sweep 3  4  7  0  4  5  6 11
down-sweep 3  0  7  4  4 11  6 16
down-sweep 0  3  4 11 11 15 16 22
reference  0  3  4 11 11 15 16 22
match

```

On a real GPU you rarely write the tree by hand. Apple GPUs have SIMD-group intrinsics, so a block scan becomes: scan inside each SIMD-group with one call, scan the eight group totals, add. [MSL](#) This kernel (not compiled in my sandbox, since there is no Metal compiler here) uses that approach; its second pass, scanning the per-block sums, is the same idea again.

Block scan using SIMD-group intrinsics

◇ MSL · not compiled here (needs Apple's Metal compiler)

```

#include <metal_stdlib>
using namespace metal;

// Exclusive prefix sum of one block of 256 values: SIMD-group scans, then a scan of the group totals.
kernel void blockScan(device const uint* in [[buffer(0)]], device uint* out [[buffer(1)]],
                      device uint* sums [[buffer(2)]], // one total per threadgroup
                      uint tid [[thread_position_in_threadgroup]], uint gid [[thread_position_in_grid]],
                      uint tgid [[threadgroup_position_in_grid]],
                      uint lane [[thread_index_in_simdgroup]],
                      uint simdIdx [[simdgroup_index_in_threadgroup]]) {
    threadgroup uint groupTotals[8]; // 256 threads / 32 lanes = 8 SIMD groups

    uint v = in[gid];
    uint within = simd_prefix_exclusive_sum(v); // scan inside my SIMD group, no memory
    uint total = simd_sum(v);                  // the group's total
    if (lane == 0) groupTotals[simdIdx] = total;
    threadgroup_barrier(mem_flags::mem_threadgroup); // make groupTotals visible

    uint before = 0; // sum of the SIMD groups to my left
    for (uint i = 0; i < simdIdx; ++i) before += groupTotals[i];
    out[gid] = before + within;
    if (tid == 255) sums[tgid] = before + within + v; // block total, scanned in a second pass
}

```

When compute beats a render pass • Use compute for anything that is not rasterize triangles”: culling, particles, blurs, histograms, building acceleration data. On Apple GPUs a compute pass cannot use tile memory the way a render pass can, so fuse post-processing into tile shaders or into the end of a render pass when you can.

Chapter 11 in one minute

- Dispatch = grid of threadgroups = SIMD-groups of 32 = threads. Threadgroups share fast memory and barriers; separate threadgroups never wait for each other.
- Avoid divergence inside a SIMD-group; choose threadgroup sizes that are multiples of 32.
- In Metal 4 you insert barriers between dependent dispatches and between compute and render work.
- Scan, reduce and compact are the building blocks; use SIMD-group intrinsics to avoid memory traffic.

Monte Carlo Rendering

How to solve the rendering equation with random numbers: why random sampling works, why smart sampling matters, and how a path tracer turns those ideas into a picture. All images are rendered by a path tracer written for this book.

12

12.1 Integrals you cannot solve by hand

Chapter 7's reflection integral has no closed form once real lights, occluders and indirect bounces are involved. **Monte Carlo integration** estimates it by averaging random samples. To estimate $\int f(x) dx$, draw random points x_i from a distribution with density $p(x)$ and average $f(x_i)/p(x_i)$:

$$I \approx \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{p(x_i)}$$

The estimate is *unbiased* (right on average) and its error falls as $1/\sqrt{N}$: four times the samples, half the noise. That slow convergence is why rendering budgets are measured in samples per pixel.

12.2 Importance sampling: aim your samples

The error depends on how much f/p varies. If p is proportional to f , the ratio is constant and one sample gives the exact answer. So you choose p to resemble the integrand, putting samples where the contribution is large. For the diffuse integral $\int \cos \theta d\omega$, sample directions with density $\cos \theta / \pi$ (**cosine-weighted**) and the cosine cancels. For a glossy lobe, sample the GGX half vector.

The listing estimates $\int_{\text{hemisphere}} \cos \theta d\omega = \pi$ with 1000 samples each way:

Uniform versus cosine-weighted sampling of the same integral

✓ compiled & run · g++13, -std=c++23 -Wall -Wextra -pedantic

```
#include <cmath>
#include <cstdio>
#include <numbers>
#include <random>

// Estimate the integral of cos(theta) over the hemisphere (exact answer: pi).
int main() {
    constexpr float pi = std::numbers::pi_v<float>;
    std::mt19937 rng(7);
    std::uniform_real_distribution<float> U(0.0f, 1.0f);
    const int N = 1000;

    for (int trial = 0; trial < 2; ++trial) {
        double sum = 0, sum2 = 0;
        for (int i = 0; i < N; ++i) {
            double f, pdf;
            if (trial == 0) {
                float cosT = U(rng); (void)U(rng); // uniform on the hemisphere
                f = cosT; pdf = 1.0 / (2 * pi); // cos(theta) is uniform in [0,1]
            } else {
                float cosT = std::sqrt(U(rng)); (void)U(rng); // cosine-weighted: pdf = cos / pi
                f = cosT; pdf = cosT / pi;
            }
            double x = f / pdf; // one sample of the estimator
            sum += x; sum2 += x * x;
        }
        double mean = sum / N, var = sum2 / N - mean * mean;
    }
}
```

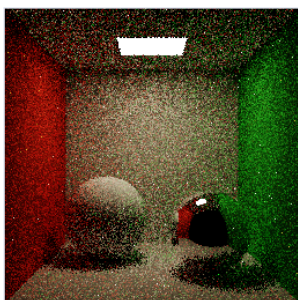
```
std::printf("%-16s estimate = %.4f  std error = %.4f\n",
            trial == 0 ? "uniform:" : "cosine-weighted:", mean, std::sqrt(var / N));
}
std::printf("exact          pi          = %.4f\n", double(pi));
}
```

OUTPUT

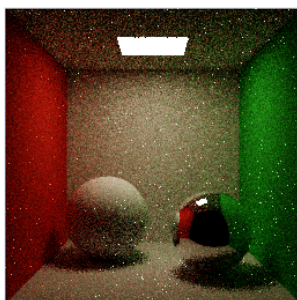
```
uniform:      estimate = 3.1234  std error = 0.0574
cosine-weighted: estimate = 3.1416  std error = 0.0000
exact        pi          = 3.1416
```

Uniform sampling gets 3.12 ± 0.06 ; cosine-weighted gets exactly π with zero error, because f/p is the constant π for every sample. Real integrands are never that kind, but the lesson transfers.

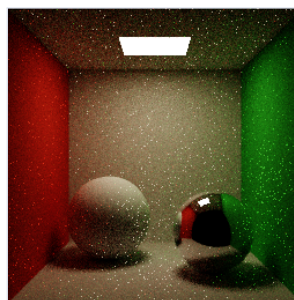
NOISE FALLS AS $1/\sqrt{N}$: ONE, FOUR, SIXTEEN AND SIXTY-FOUR SAMPLES PER PIXEL



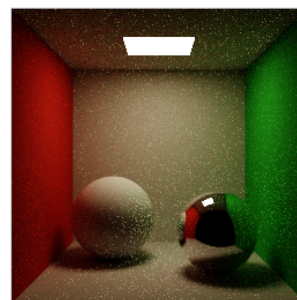
1 spp



4 spp

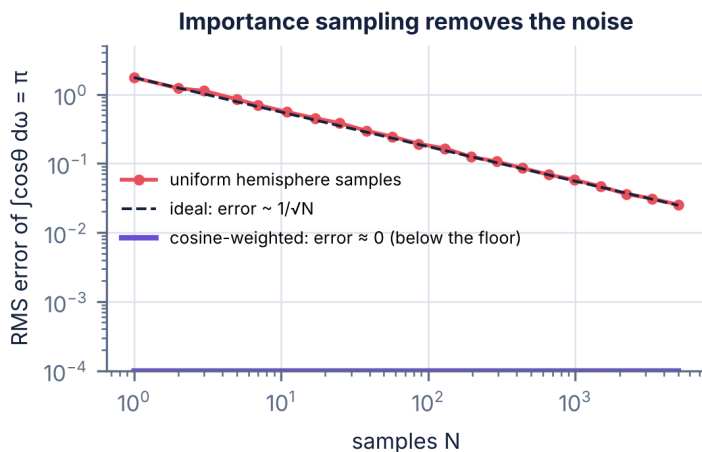


16 spp

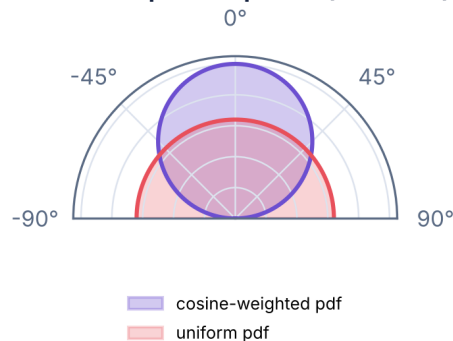


64 spp

ERROR VS. SAMPLES (LEFT) AND WHERE EACH STRATEGY PUTS ITS SAMPLES (RIGHT), PLOTTED FROM THE GUIDE'S OWN MONTE CARLO CODE



Where samples are placed (side view)



Uniform sampling follows the $1/\sqrt{N}$ line: $100\times$ the samples for $10\times$ less error. Cosine-weighted sampling puts more directions where the cosine is large, so for this particular integrand the error vanishes. Engines combine modest sample counts with **denoisers** and temporal accumulation.

12.3 Random numbers and sampling helpers on the GPU

Each GPU thread needs its own random stream. The listing uses a PCG hash seeded from the pixel and frame index, an orthonormal basis builder to rotate a local sample into the world, and the cosine-weighted hemisphere mapping. These functions ran through the CPU shim and produced the pictures in this chapter.

Hash RNG, basis construction and cosine-weighted hemisphere sampling

✓ MSL - identical to code compiled & run on the CPU through a shim

```
// ---- PCG hash: a tiny, high-quality per-thread random number generator ----
inline uint pcgHash(uint v) {
    uint state = v * 747796405u + 2891336453u;
    uint word = ((state >> ((state >> 28u) + 4u)) ^ state) * 277803737u;
    return (word >> 22u) ^ word;
}

inline float rand01(thread uint& state) { // uniform in [0, 1)
    state = pcgHash(state);
    return float(state >> 8) * (1.0f / 16777216.0f);
}

// ---- build an orthonormal basis around a unit normal (Duff et al., branchless) ----
inline void buildBasis(float3 n, thread float3& t, thread float3& b) {
    float s = n.z >= 0.0f ? 1.0f : -1.0f;
    float a = -1.0f / (s + n.z);
    float c = n.x * n.y * a;
    t = float3(1.0f + s * n.x * n.x * a, s * c, -s * n.x);
    b = float3(c, s + n.y * n.y * a, -n.y);
}

// ---- cosine-weighted hemisphere sample: pdf = cos(theta) / pi ----
inline float3 cosineSampleHemisphere(float2 u) {
    float r = sqrt(u.x);
    float phi = 2.0f * 3.14159265358979f * u.y;
    return float3(r * cos(phi), r * sin(phi), sqrt(max(0.0f, 1.0f - u.x))); // z is "up" (the normal)
}

inline float3 toWorld(float3 local, float3 n) {
    float3 t, b;
    buildBasis(n, t, b);
    return t * local.x + b * local.y + n * local.z;
}
```

Importance-sampling the GGX half vector

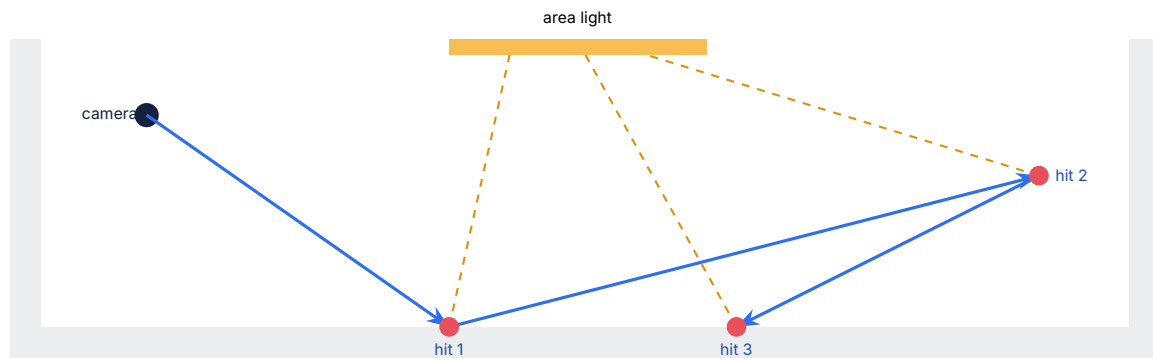
✓ MSL - identical to code compiled & run on the CPU through a shim

```
// ---- GGX importance sampling of the half vector (local space, z = normal) ----
inline float3 sampleGGXHalf(float2 u, float alpha) {
    float phi = 2.0f * 3.14159265358979f * u.x;
    float cosTheta = sqrt((1.0f - u.y) / (1.0f + (alpha * alpha - 1.0f) * u.y));
    float sinTheta = sqrt(max(0.0f, 1.0f - cosTheta * cosTheta));
    return float3(sinTheta * cos(phi), sinTheta * sin(phi), cosTheta);
}
```

12.4 From integral to path tracer

A **path tracer** applies Monte Carlo to the full rendering equation. For each pixel: shoot a ray into the scene, find the surface, *estimate direct light* (pick a point on a light, shoot a shadow ray), then pick a random bounce direction by sampling the BRDF and continue from the new hit. Each bounce adds one more order of indirect light.

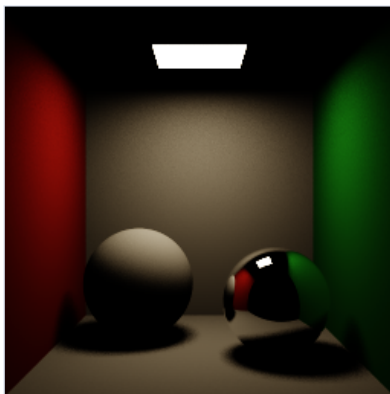
ONE PATH: CAMERA RAY, TWO BOUNCES, A SHADOW RAY AT EVERY HIT (NEXT-EVENT ESTIMATION")



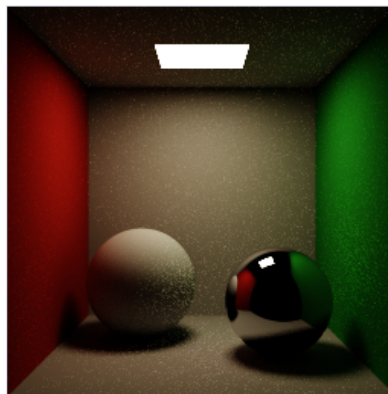
Solid: the path. **Dashed:** shadow rays to the light.

Each hit adds $(\text{BRDF} \times \text{light} \times \text{cosine})$ if the shadow ray is clear, then the path continues.

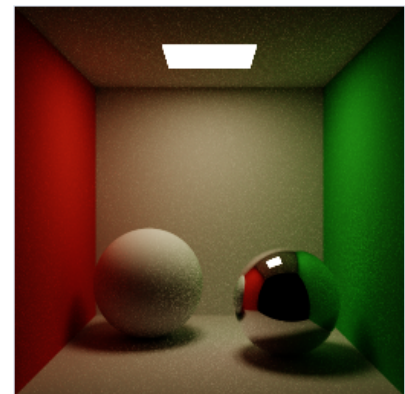
BUILDING UP GLOBAL ILLUMINATION IN A CORNELL BOX: DIRECT LIGHT, ONE INDIRECT BOUNCE, FULL PATHS



direct light only: black shadows



plus one bounce: color bleeding begins



full paths (6 bounces, 512 spp)

Look at the walls in the middle image: the red and green walls tint the neighboring white surfaces and ceiling. That **color bleeding**, and the soft darkening of corners, is global illumination. A rasterizer needs special-purpose tricks to fake it; a ray tracer gets it by following light.

Path tracer ingredients checklist

Ingredient	What it does
Next-event estimation	Direct shadow ray to a light at each hit: far less noise than hoping a random bounce finds the light
Importance-sampled BRDF	Directions chosen where the BRDF is large (cosine for diffuse, GGX half vector for glossy)
Russian roulette	Randomly end low-contribution paths, dividing survivors' weight: unbiased, shorter average path
Multiple importance sampling	Combine light sampling and BRDF sampling with weights, for the best of both
Denoising + accumulation	Turns 1–4 spp noisy output into a clean image; the key to real-time

Chapter 12 in one minute

- Monte Carlo averages f/p over random samples; unbiased, error $\propto 1/\sqrt{N}$.
- Importance sampling picks $p \propto f$: cosine-weighted for diffuse, GGX half vector for glossy.
- A path tracer shoots camera rays, adds direct light via shadow rays at each hit, and bounces by sampling the BRDF.
- Real-time engines run few samples and rely on denoisers and temporal accumulation.

Ray Tracing

Rays versus triangles, why a BVH makes tracing fast, how Metal's hardware ray tracing exposes it, and the three effects you will build first: shadows, ambient occlusion and global illumination.

13

13.1 A ray and a triangle

A ray is $\mathbf{o} + t\mathbf{d}$ for $t > 0$. To find where it hits a triangle, use the **Möller-Trumbore** algorithm, which solves for the barycentric coordinates (u, v) and the distance t at once with a few cross and dot products. Chapter 5's barycentric weights come back: a hit is valid when $u, v \geq 0$, $u + v \leq 1$ and $t > 0$.

Ray-triangle (Möller-Trumbore) and ray-box (slab) tests

✓ compiled & run · together with the next listing

```
#include <algorithm>
#include <array>
#include <cmath>
#include <cstdio>
#include <span>
#include <vector>

struct V3 {
    float x, y, z;
};

constexpr V3 operator-(V3 a, V3 b) { return {a.x - b.x, a.y - b.y, a.z - b.z}; }
constexpr float dot(V3 a, V3 b) { return a.x * b.x + a.y * b.y + a.z * b.z; }
constexpr V3 cross(V3 a, V3 b) {
    return {a.y * b.z - a.z * b.y, a.z * b.x - a.x * b.z, a.x * b.y - a.y * b.x};
}
constexpr float at(V3 v, int i) { return i == 0 ? v.x : (i == 1 ? v.y : v.z); }

struct Tri {
    V3 a, b, c;
};

// Moller-Trumbore ray/triangle test. Returns distance t, or a negative number for a miss.
float hitTriangle(V3 o, V3 d, const Tri& t) {
    V3 e1 = t.b - t.a, e2 = t.c - t.a;
    V3 p = cross(d, e2);
    float det = dot(e1, p);
    if (std::fabs(det) < 1e-8f) return -1; // ray parallel to the triangle
    float inv = 1.0f / det;
    V3 s = o - t.a;
    float u = dot(s, p) * inv;
    if (u < 0 || u > 1) return -1;
    V3 q = cross(s, e1);
    float v = dot(d, q) * inv;
    if (v < 0 || u + v > 1) return -1;
    float dist = dot(e2, q) * inv;
    return dist > 1e-4f ? dist : -1;
}

struct Aabb {
    V3 lo{1e30f, 1e30f, 1e30f}, hi{-1e30f, -1e30f, -1e30f};
};

void grow(Aabb& b, V3 p) {
    b.lo = {std::min(b.lo.x, p.x), std::min(b.lo.y, p.y), std::min(b.lo.z, p.z)};
    b.hi = {std::max(b.hi.x, p.x), std::max(b.hi.y, p.y), std::max(b.hi.z, p.z)};
}

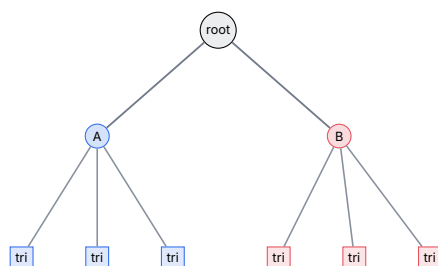
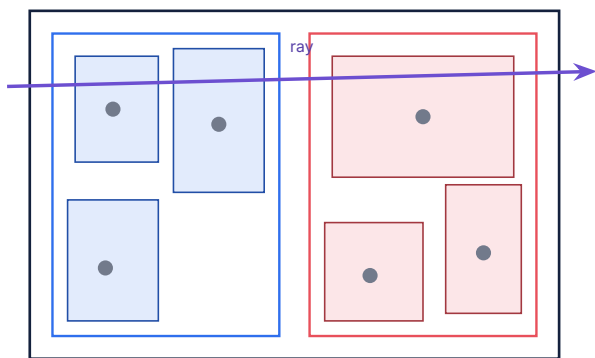
// Slab test: does the ray enter the box before tMax?
bool hitBox(const Aabb& b, V3 o, V3 invD, float tMax) {
```

```
float t0 = 0, t1 = tMax;
for (int i = 0; i < 3; ++i) {
    float a = (at(b.lo, i) - at(o, i)) * at(invD, i), c = (at(b.hi, i) - at(o, i)) * at(invD, i);
    t0 = std::max(t0, std::min(a, c));
    t1 = std::min(t1, std::max(a, c));
}
return t0 <= t1;
}
```

13.2 Why you need an acceleration structure

A scene may have millions of triangles and every pixel shoots several rays. Testing every triangle per ray is hopeless. A **bounding volume hierarchy** (BVH) wraps groups of triangles in axis-aligned boxes, then groups of boxes in bigger boxes, forming a tree. A ray that misses a box skips everything inside.

A BVH: BOXES NESTED IN BOXES. THE RAY (ARROW) SKIPS WHOLE SUBTREES ITS PATH NEVER ENTERS



Visit root, test the child boxes, descend only into boxes the ray enters.

A median-split BVH with stack traversal, tested against brute force

✓ compiled & run - g++ 13, -std=c++23 -Wall -Wextra -pedantic

```
struct Node {
    Aabb box;
    int left = -1, right = -1, first = 0, count = 0;
}; // leaf when left < 0

struct Bvh {
    std::vector<Tri> tris;
    std::vector<Node> nodes;
    explicit Bvh(std::vector<Tri> t) : tris(std::move(t)) {
        nodes.reserve(2 * tris.size());
        build(0, int(tris.size()));
    }
    static V3 centroid(const Tri& t) {
        return {(t.a.x + t.b.x + t.c.x) / 3, (t.a.y + t.b.y + t.c.y) / 3, (t.a.z + t.b.z + t.c.z) / 3};
    }
    int build(int first, int count) {
        int id = int(nodes.size());
        nodes.emplace_back();
        Aabb box, cb;
        for (int i = first; i < first + count; ++i) {
            grow(box, tris[i].a);
            grow(box, tris[i].b);
            grow(box, tris[i].c);
            grow(cb, centroid(tris[i]));
        }
        nodes[id].box = box;
        if (count <= 4) {
            nodes[id].first = first;
            nodes[id].count = count;
            return id;
        }
    }
}
```

```

V3 ext = cb.hi - cb.lo;
int axis =
    (ext.x > ext.y && ext.x > ext.z) ? 0 : (ext.y > ext.z ? 1 : 2); // split the widest axis
int mid = first + count / 2;
std::nth_element(tris.begin() + first, tris.begin() + mid, tris.begin() + first + count,
    [axis](const Tri& a, const Tri& b) {
        return at(centroid(a), axis) < at(centroid(b), axis);
    });
int l = build(first, mid - first), r = build(mid, first + count - mid);
nodes[id].left = l;
nodes[id].right = r;
return id;
}

// Closest hit; counts the work done so we can compare with brute force.
float trace(V3 o, V3 d, int& boxTests, int& triTests) const {
    V3 invD{1 / d.x, 1 / d.y, 1 / d.z};
    float best = 1e30f;
    std::array<int, 64> stack{};
    int sp = 0;
    stack[sp++] = 0;
    while (sp > 0) {
        const Node& n = nodes[stack[--sp]];
        ++boxTests;
        if (!hitBox(n.box, o, invD, best)) continue;
        if (n.left < 0) {
            for (int i = n.first; i < n.first + n.count; ++i) {
                ++triTests;
                float t = hitTriangle(o, d, tris[i]);
                if (t > 0 && t < best) best = t;
            }
        } else {
            stack[sp++] = n.left;
            stack[sp++] = n.right;
        }
    }
    return best;
}

// A bumpy 64x64 height field = 8192 triangles.
std::vector<Tri> heightField(int n) {
    auto h = [(float x, float z) {
        return 0.35f * std::sin(3 * x) * std::cos(2.5f * z) + 0.1f * std::sin(9 * x + z);
    }];
    std::vector<Tri> out;
    for (int j = 0; j < n; ++j)
        for (int i = 0; i < n; ++i) {
            float x0 = -2 + 4.0f * i / n, x1 = -2 + 4.0f * (i + 1) / n, z0 = -2 + 4.0f * j / n,
                z1 = -2 + 4.0f * (j + 1) / n;
            V3 a{x0, h(x0, z0), z0}, b{x1, h(x1, z0), z0}, c{x1, h(x1, z1), z1}, d{x0, h(x0, z1), z1};
            out.push_back({a, b, c});
            out.push_back({a, c, d});
        }
    return out;
}

int main() {
    Bvh bvh(heightField(64));
    long boxes = 0, tris = 0;
    int hits = 0;
    const int N = 64;
    for (int y = 0; y < N; ++y)
        for (int x = 0; x < N; ++x) {
            V3 o{-1.9f + 3.8f * x / N, 3.0f, -1.9f + 3.8f * y / N}, d{0, -1, 0};
            int b = 0, t = 0;
            hits += bvh.trace(o, d, b, t) < 1e29f;
            boxes += b;
            tris += t;
        }
    std::printf("triangles: %zu, BVH nodes: %zu\n", bvh.tris.size(), bvh.nodes.size());
    std::printf("rays: %d, hits: %d\n", N * N, hits);
    std::printf(
        "per ray: brute force = %zu triangle tests; BVH = %.1f box tests + %.1f triangle tests\n",

```

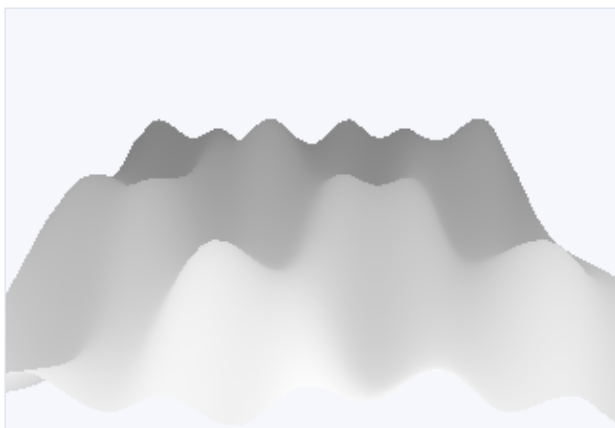
```
    bvh.tris.size(), double(boxes) / (N * N), double(tris) / (N * N));
}
```

OUTPUT

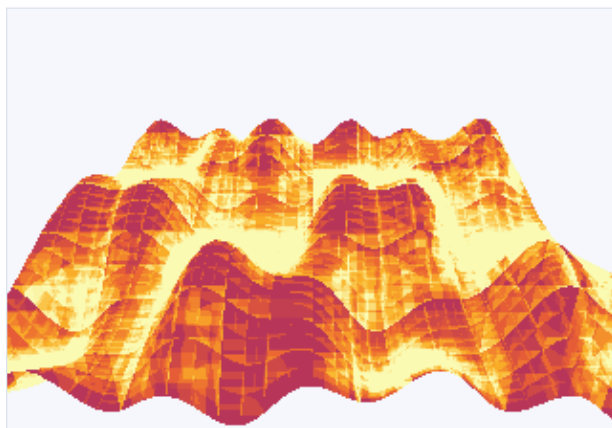
```
triangles: 8192, BVH nodes: 4095
rays: 4096, hits: 4096
per ray: brute force = 8192 triangle tests; BVH = 24.3 box tests + 5.3 triangle tests
```

On the 8192-triangle bumpy terrain, a ray needs **24 box tests and 5 triangle tests** on average instead of 8192 triangle tests: roughly 280 times fewer. The images show what this costs spatially: the heat map colors each pixel by how many nodes its ray visited.

THE TERRAIN SHADED (LEFT) AND THE NUMBER OF BVH NODES EACH RAY VISITED (RIGHT)



depth-shaded terrain, 8192 triangles

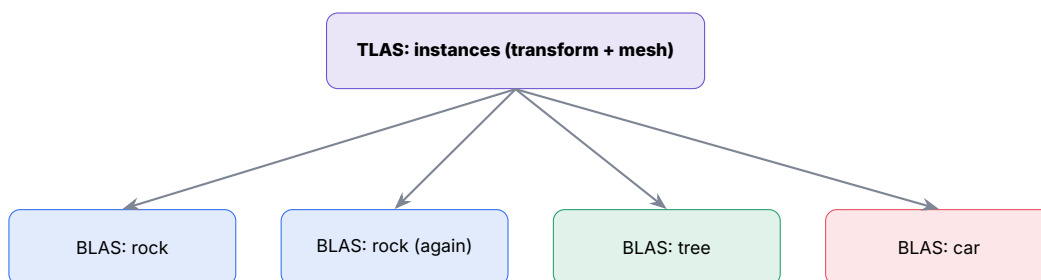


traversal cost per pixel (brighter = more nodes visited)

13.3 Hardware ray tracing on Apple GPUs

M3-class GPUs and later contain ray-tracing hardware that traverses acceleration structures for you. You *do not* write the traversal loop above; you build the structures with Metal and ask the intersector. Use `supportsRaytracing()` to check (chapter 9's capability listing).

TWO LEVELS: ONE BLAS PER MESH, ONE TLAS FOR THE SCENE



A **BLAS** (bottom-level) holds one mesh's triangles; build it once, rebuild or refit only when the mesh deforms. The **TLAS** (top-level) lists instances, each a transform plus a reference to a BLAS, so moving an object is a cheap TLAS update, and many instances can share one BLAS.

Metal 4 The listing builds a BLAS from an indexed triangle buffer and a TLAS from an instance buffer. In Metal 4 an instance refers to its BLAS by `ResourceID`, which lets a compute shader fill the instance buffer on the GPU. Two details worth a comment: the **scratch buffer** is memory the builder needs only during the build, and every acceleration structure and scratch buffer must be resident (chapter 9).

Build a BLAS and a TLAS

✓ type-checked against Apple's real metal-cpp headers · never run on a GPU

```

#include <Metal/MTL4AccelerationStructure.hpp> // not pulled in by Metal.hpp in this metal-cpp snapshot

struct Accel {
    MTL::AccelerationStructure* as;
    MTL::Buffer* scratch;
};

// Bottom-level acceleration structure: one triangle mesh. Encode this on a compute encoder.
Accel buildBLAS(Renderer& r, MTL4::ComputeCommandEncoder* ce, MTL::Buffer* positions,
               MTL::Buffer* indices, uint32_t triangleCount) {
    auto* tri = MTL4::AccelerationStructureTriangleGeometryDescriptor::alloc()->init();
    tri->setVertexBuffer(MTL4::BufferRange::Make(positions->gpuAddress(), positions->length()));
    tri->setVertexFormat(MTL::AttributeFormatFloat3);
    tri->setVertexStride(3 * sizeof(float)); // tightly packed float3 positions
    tri->setIndexBuffer(MTL4::BufferRange::Make(indices->gpuAddress(), indices->length()));
    tri->setIndexType(MTL::IndexTypeUInt32);
    tri->setTriangleCount(triangleCount);
    tri->setOpaque(true); // no any-hit shader: faster traversal

    auto* desc = MTL4::PrimitiveAccelerationStructureDescriptor::alloc()->init();
    desc->setGeometryDescriptors(NS::Array::array(tri));

    MTL::AccelerationStructureSizes sizes = r.device->accelerationStructureSizes(desc);
    Accel a;
    a.as = r.device->newAccelerationStructure(sizes.accelerationStructureSize);
    a.scratch = r.device->newBuffer(sizes.buildScratchBufferSize, MTL::ResourceStorageModePrivate);
    r.residency->addAllocation(a.as);
    r.residency->addAllocation(a.scratch);
    r.residency->commit();

    ce->buildAccelerationStructure(
        a.as, desc, MTL4::BufferRange::Make(a.scratch->gpuAddress(), a.scratch->length()));
    desc->release();
    tri->release();
    return a;
}

// Top-level structure: instances = (transform, which BLAS). In Metal 4 an instance names its BLAS
// by ResourceID, so the instance buffer can be filled by the CPU or by a compute shader.
Accel buildTLAS(Renderer& r, MTL4::ComputeCommandEncoder* ce, MTL::Buffer* instanceBuffer,
               uint32_t instanceCount) {
    auto* desc = MTL4::InstanceAccelerationStructureDescriptor::alloc()->init();
    desc->setInstanceCount(instanceCount);
    desc->setInstanceDescriptorType(MTL::AccelerationStructureInstanceDescriptorTypeIndirect);
    desc->setInstanceDescriptorStride(sizeof(MTL::IndirectAccelerationStructureInstanceDescriptor));
    desc->setInstanceTransformationMatrixLayout(MTL::MatrixLayoutColumnMajor);
    desc->setInstanceDescriptorBuffer(
        MTL4::BufferRange::Make(instanceBuffer->gpuAddress(), instanceBuffer->length()));

    MTL::AccelerationStructureSizes sizes = r.device->accelerationStructureSizes(desc);
    Accel a;
    a.as = r.device->newAccelerationStructure(sizes.accelerationStructureSize);
    a.scratch = r.device->newBuffer(sizes.buildScratchBufferSize, MTL::ResourceStorageModePrivate);
    r.residency->addAllocation(a.as);
    r.residency->addAllocation(a.scratch);
    r.residency->commit();

    // The TLAS reads the BLAS, so the BLAS builds must be finished first.
    ce->barrierAfterEncoderStages(MTL::StageAccelerationStructure, MTL::StageAccelerationStructure,
                                MTL4::VisibilityOptionDevice);
    ce->buildAccelerationStructure(
        a.as, desc, MTL4::BufferRange::Make(a.scratch->gpuAddress(), a.scratch->length()));
    desc->release();
    return a;
}

void fillInstance(MTL::IndirectAccelerationStructureInstanceDescriptor& out,
                MTL::AccelerationStructure* blas, const float rowMajor3x4[12]) {
    for (int col = 0; col < 4; ++col) // PackedFloat4x3 = 4 columns of 3
        out.transformationMatrix.columns[col] = MTL::PackedFloat3(
            rowMajor3x4[0 * 4 + col], rowMajor3x4[1 * 4 + col], rowMajor3x4[2 * 4 + col]);
    out.options = MTL::AccelerationStructureInstanceOptionOpaque;
    out.mask = 0xFF;
}

```

```

out.intersectionFunctionTableOffset = 0;
out.userID = 0;
out.accelerationStructureID = blas->gpuResourceID();
}

```

13.4 Tracing from a compute shader: soft shadows

MSL This kernel reconstructs each pixel's world position from the depth buffer, picks a random point on a disc light, and shoots one shadow ray. A shadow ray only needs to know *whether* anything is in the way, so it asks the intersector to accept the first hit (`accept_any_intersection`), which is cheaper than finding the nearest. It reuses the random and basis helpers from chapter 12.

Shadow-ray compute kernel ◇ MSL · not compiled here (needs Apple's Metal compiler)

```

#include <metal_stdlib>
#include <metal_raytracing>
using namespace metal;
using namespace metal::raytracing;

struct Frame {
    float4x4 invViewProj;           // clip -> world
    float4 lightPos;                // xyz = center of a disc area light, w = radius
    uint2 size;
    uint frameIndex;
};

kernel void shadowRays(instance_acceleration_structure scene [[buffer(0)]],
                      constant Frame& f [[buffer(1)]],
                      depth2d<float, access::read> depthTex [[texture(0)]],
                      texture2d<float, access::read> normalTex [[texture(1)]],
                      texture2d<float, access::write> shadowTex [[texture(2)]],
                      uint2 gid [[thread_position_in_grid]]) {
    if (gid.x >= f.size.x || gid.y >= f.size.y) return;

    // 1. Rebuild the world-space position from the depth buffer.
    float2 uv = (float2(gid) + 0.5) / float2(f.size);
    float4 clip = float4(uv.x * 2 - 1, 1 - uv.y * 2, depthTex.read(gid), 1);
    float4 w = f.invViewProj * clip;
    float3 P = w.xyz / w.w;
    float3 N = normalize(normalTex.read(gid).xyz * 2 - 1);

    // 2. Pick a random point on the light's disc (soft shadows = many such samples over frames).
    uint rng = pcgHash(gid.x + gid.y * f.size.x + f.frameIndex * 9781u);
    float2 u2 = float2(rand01(rng), rand01(rng));
    float3 t, b; buildBasis(normalize(f.lightPos.xyz - P), t, b);
    float r = f.lightPos.w * sqrt(u2.x), phi = 6.2831853 * u2.y;
    float3 target = f.lightPos.xyz + (t * cos(phi) + b * sin(phi)) * r;
    float3 L = target - P;
    float dist = length(L);

    // 3. Shadow ray: any hit at all means "in shadow", so stop at the first one.
    ray ray;
    ray.origin = P + N * 1e-3; // bias off the surface
    ray.direction = L / dist;
    ray.min_distance = 0.0;
    ray.max_distance = dist - 1e-3;

    intersector<instancing> isect;
    isect.accept_any_intersection(true);
    auto hit = isect.intersect(ray, scene);
    shadowTex.write(hit.type == intersection_type::none ? 1.0 : 0.0, gid);
}

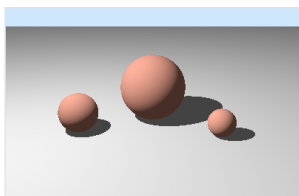
```

Binding and dispatching it, with barriers

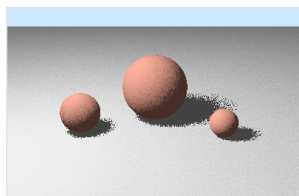
✓ type-checked against Apple's real metal-cpp headers · never run on a GPU

```
// Trace one shadow ray per pixel in a compute kernel (the kernel is shown in the MSL listing).
void encodeShadowRays(Renderer& r, MTL4::ComputeCommandEncoder* ce, MTL::ComputePipelineState* pso,
    MTL::AccelerationStructure* tlas, MTL::Texture* depth, MTL::Texture* normals,
    MTL::Texture* shadowMask, MTL::GPUAddress frameConstants) {
    r.args->setResource(tlas->gpuResourceID(), 0); // [[buffer(0)]] instance_acceleration_structure
    r.args->setTexture(depth->gpuResourceID(), 0); // [[texture(0)]]
    r.args->setTexture(normals->gpuResourceID(), 1);
    r.args->setTexture(shadowMask->gpuResourceID(), 2);
    r.args->setAddress(frameConstants, 1); // [[buffer(1)]]

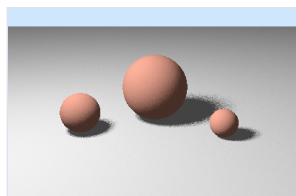
    ce->setArgumentTable(r.args);
    ce->setComputePipelineState(pso);
    // G-buffer must be finished (fragment) before the dispatch reads it; TLAS build before tracing.
    ce->barrierAfterQueueStages(MTL::StageFragment | MTL::StageAccelerationStructure,
        MTL::StageDispatch, MTL4::VisibilityOptionDevice);
    ce->dispatchThreads(MTL::Size::Make(shadowMask->width(), shadowMask->height(), 1),
        MTL::Size::Make(8, 8, 1));
}
```

HARD VERSUS SOFT SHADOWS: ONE SAMPLE IS NOISY, AVERAGING CONVERGES (RENDERED BY THE GUIDE'S CPU RAY TRACER)

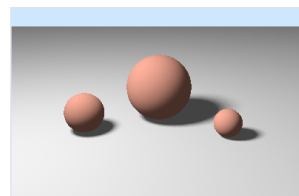
point light: hard edge



area light, 1 ray



8 rays

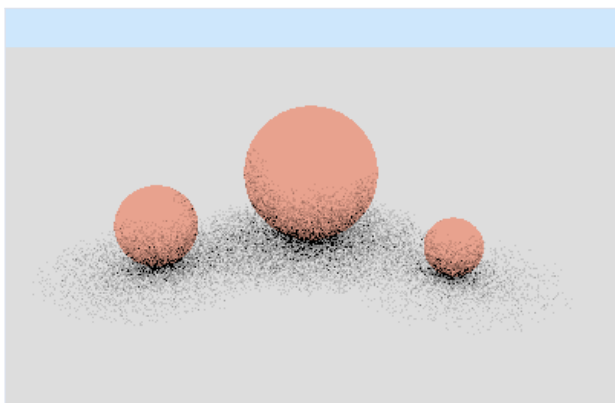


256 rays

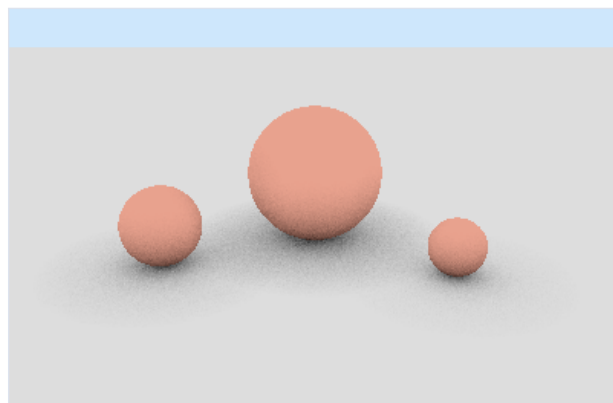
A point light gives razor-sharp shadows; real lights have area, so shadows have a soft **penumbra**. One random ray per pixel per frame gives a noisy mask; accumulating over frames or denoising turns it into a smooth one. That is the main design pattern of real-time ray tracing: *few rays, then filter*.

13.5 Ambient occlusion and global illumination

Ambient occlusion (AO) asks how much of the hemisphere above a point is blocked within a short distance, darkening creases. With rays: shoot a few cosine-weighted rays with a short maximum distance and count the fraction that escape. **Global illumination** (GI) goes further and, when a ray hits something, *also* reads that surface's lighting: one extra bounce gives you color bleeding (chapter 12's middle image).

AMBIENT OCCLUSION FROM RAYS: 4 RAYS PER PIXEL (NOISY) VERSUS 128 (CONVERGED)

4 rays per pixel



128 rays per pixel

A practical hybrid frame (the design this guide recommends for an engine like yours)

Pass	Technique	Notes
Primary visibility	Rasterize G-buffer	Cheap and exact; Apple GPUs excel at it
Shadows	1 ray/pixel to a light sample	Denoise with temporal accumulation
Ambient / GI	1 cosine-weighted ray, one bounce	Reads a cached or lit surface; denoise heavily
Reflections	Rays for smooth surfaces only	Fall back to a probe or screen-space for rough ones
Final	Light, tone map, encode	Chapter 7's order

Pitfalls • Self-intersection: a ray starting exactly on a surface re-hits it. Offset the origin along the normal (the listing uses 10^{-3}) or set a minimum distance. **Residency and barriers:** the BLAS build must finish before the TLAS build, and both before any trace (see the dispatch listing's barrier). **Divergence:** neighboring rays hitting different materials run slowly; sort or bin when it matters.

Chapter 13 in one minute

- Möller–Trumbore intersects a ray with a triangle; a BVH reduces thousands of tests per ray to a few dozen.
- Metal: build BLASes (one per mesh) and a TLAS (instances); trace from compute (or render) with the intersector; keep everything resident and add barriers.
- Shadow rays use any-hit, one jittered ray per pixel per frame, then denoise. AO and GI use cosine-weighted rays.
- Hybrid rendering: rasterize visibility, ray trace shadows and light, filter.

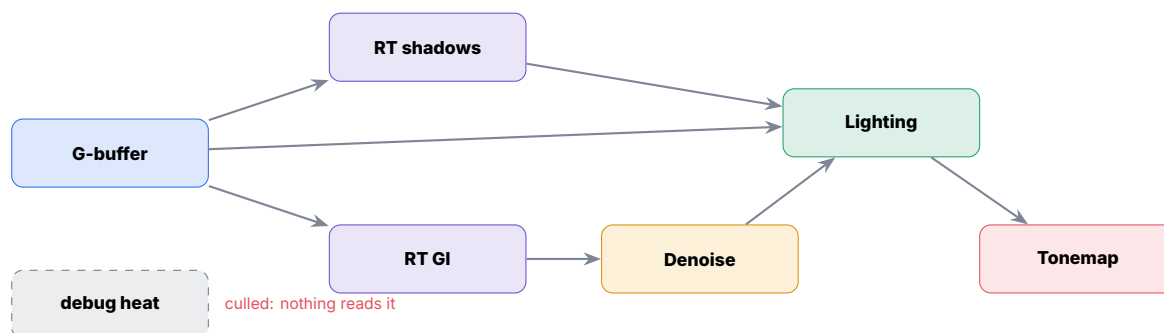
The Render Graph

A modern frame is a dozen passes passing images to each other. A render graph declares who reads and writes what, then works out the order, the barriers and the memory sharing for you. This chapter builds one that runs.

14.1 The problem

Chapter 9 said Metal 4 leaves barriers and lifetimes to you. In a frame with a G-buffer, ray-traced shadows, ray-traced GI, a denoiser, lighting, tone mapping and debug views, you will either hand-write dozens of barriers (and get some wrong) or build a system that derives them. That system is the **render graph**.

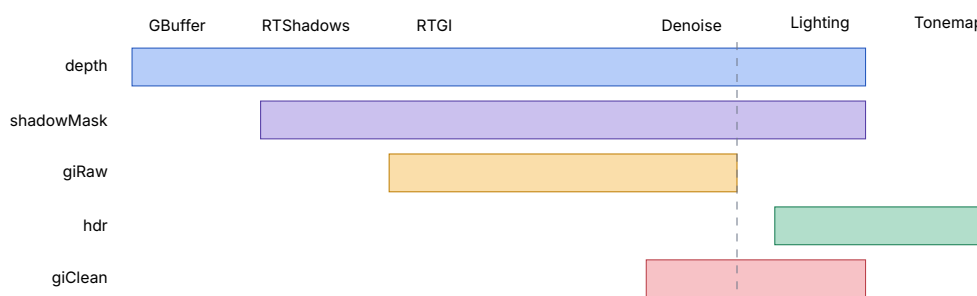
ONE FRAME AS A GRAPH: PASSES ARE NODES, RESOURCES ARE THE EDGES BETWEEN THEM



14.2 What the graph does

1. **Cull** passes whose outputs nobody uses (the dashed pass above), walking backwards from passes with side effects like presenting.
2. **Order** passes and derive the **barriers**: a pass that reads a resource waits for the pass that last wrote it.
3. **Alias memory**: resources whose lifetimes do not overlap share one allocation, which on a big frame saves hundreds of megabytes.

MEMORY ALIASING: giRaw "DIES BEFORE" hdr "IS BORN", SO THEY SHARE A SLOT (FROM THE LISTING'S OUTPUT)



giRaw (passes 2–4) and hdr (5–6) never exist at the same time: one allocation serves both.

14.3 A working render graph

The listing is a complete, small render graph on the CPU. It is written like the real thing but runs without a GPU, so you can see its decisions. Resources are named, passes declare reads and writes, `compile()` culls, derives barriers and aliases transient resources. The output below is the program's actual output.

Culling, barrier derivation and memory aliasing for a hybrid frame

✓ compiled & run: g++ 13, -std=c++23 -Wall -Wextra -pedantic

```
#include <algorithm>
#include <cstdio>
#include <string>
#include <vector>

struct Pass {
    std::string name;
    std::vector<int> reads, writes;
    bool sideEffect = false; // e.g. present: must run even if nothing reads it
};

class RenderGraph {
public:
    int resource(std::string name,
                bool transient = true) { // imported = owned outside (swapchain, history)
        names_.push_back(std::move(name));
        transient_.push_back(transient);
        return int(names_.size()) - 1;
    }
    void pass(std::string name, std::vector<int> reads, std::vector<int> writes,
             bool sideEffect = false) {
        passes_.push_back({std::move(name), std::move(reads), std::move(writes), sideEffect});
    }

    void compile() const {
        const int P = int(passes_.size()), R = int(names_.size());
        // 1. Cull: walk backwards from side-effect passes, keeping every pass that feeds them.
        std::vector<bool> live(P, false), needed(R, false);
        for (int p = P - 1; p >= 0; --p) {
            bool feeds = passes_[p].sideEffect;
            for (int w : passes_[p].writes) feeds = feeds || needed[w];
            if (!feeds) continue;
            live[p] = true;
            for (int r : passes_[p].reads) needed[r] = true;
        }
        // 2. Barriers: a live pass that reads a resource waits for the live pass that last wrote it.
        std::vector<int> lastWriter(R, -1), first(R, -1), last(R, -1);
        std::puts("schedule:");
        for (int p = 0; p < P; ++p) {
            if (!live[p]) {
                std::printf("  -- %-10s CULLED (nothing reads its output)\n", passes_[p].name.c_str());
                continue;
            }
            std::printf("    %2d %-10s", p, passes_[p].name.c_str());
            std::vector<int> waits; // one barrier per distinct producer pass
            for (int r : passes_[p].reads)
                if (lastWriter[r] >= 0 && std::ranges::find(waits, lastWriter[r]) == waits.end())
                    waits.push_back(lastWriter[r]);
            if (!waits.empty()) std::printf("  barrier after:");
            for (int w : waits) std::printf(" %s", passes_[w].name.c_str());
            std::puts("");
            for (int r : passes_[p].reads) {
                if (first[r] < 0) first[r] = p;
                last[r] = p;
            }
            for (int w : passes_[p].writes) {
                if (first[w] < 0) first[w] = p;
                last[w] = p;
                lastWriter[w] = p;
            }
        }
        // 3. Alias: resources whose lifetimes do not overlap can share one allocation.
        std::vector<int> slotFreeAfter; // per slot: last pass index that uses it
        std::puts("memory slots:");
    }
};
```

```

std::vector<std::vector<std::string>> slots;
for (int r = 0; r < R; ++r) {
    if (first[r] < 0 || !transient_[r]) continue;
    int s = 0;
    while (s < int(slots.size()) && slotFreeAfter[s] >= first[r]) ++s;
    if (s == int(slots.size())) {
        slots.emplace_back();
        slotFreeAfter.push_back(-1);
    }
    slots[s].push_back(names_[r] + "[" + std::to_string(first[r]) + ".." +
        std::to_string(last[r]) + "]");
    slotFreeAfter[s] = last[r];
}
for (size_t s = 0; s < slots.size(); ++s) {
    std::printf(" slot %zu:", s);
    for (auto& n : slots[s]) std::printf(" %s", n.c_str());
    std::puts("");
}
}

private:
std::vector<std::string> names_;
std::vector<bool> transient_;
std::vector<Pass> passes_;
};

int main() {
    RenderGraph g;
    int depth = g.resource("depth"), nrm = g.resource("normals"), alb = g.resource("albedo");
    int shadow = g.resource("shadowMask"), gi = g.resource("giRaw"), giOk = g.resource("giClean");
    int hdr = g.resource("hdr"), heat = g.resource("bvhHeat");
    int back = g.resource("backbuffer", false); // imported: the swapchain owns it
    g.pass("GBuffer", {}, {depth, nrm, alb});
    g.pass("RTShadows", {depth, nrm}, {shadow});
    g.pass("RTGI", {depth, nrm}, {gi});
    g.pass("DebugHeat", {depth}, {heat}); // debug view nobody asked for
    g.pass("Denoise", {gi, depth, nrm}, {giOk});
    g.pass("Lighting", {depth, nrm, alb, shadow, giOk}, {hdr});
    g.pass("Tonemap", {hdr}, {back});
    g.pass("Present", {back}, {}, true);
    g.compile();
}

```

```

OUTPUT
schedule:
0  GBuffer
1  RTShadows  barrier after: GBuffer
2  RTGI       barrier after: GBuffer
-- DebugHeat  CULLED (nothing reads its output)
4  Denoise    barrier after: RTGI GBuffer
5  Lighting   barrier after: GBuffer RTShadows Denoise
6  Tonemap    barrier after: Lighting
7  Present    barrier after: Tonemap
memory slots:
slot 0: depth[0..5]
slot 1: normals[0..5]
slot 2: albedo[0..5]
slot 3: shadowMask[1..5]
slot 4: giRaw[2..4] hdr[5..6]
slot 5: giClean[4..5]

```

Read the output: **DebugHeat** is culled because nothing reads it; the Denoise pass waits on RTGI and the G-buffer; and giRaw and hdr share a memory slot. Imported resources (the swapchain's back buffer) are excluded from aliasing because something else owns them.

Wiring it to Metal 4 • In an engine, each Pass also carries a lambda that records commands. After `compile()`, execution walks the live passes in order: emit the barriers computed for it with `barrierAfterQueueStages` (using the stage masks of the producer and the consumer), open the encoder type the pass needs, call the lambda. Aliased resources come from `MTL::Heap` allocations; in Metal 4 you also add the heap to the residency set.

14.4 The G-buffer and debug views

A **G-buffer** stores per-pixel surface data (normal, albedo, roughness, metalness, depth, an object ID) in several render targets so later passes can light it without re-rasterizing. On Apple GPUs, the data stays in tile memory when the lighting runs in the same render pass, which is where TBDR shines (chapter 1). The listing writes the G-buffer and provides a debug resolve that picks a view with one integer.

G-buffer output and a switchable debug-view resolve

◇ MSL · not compiled here (needs Apple's Metal compiler)

```
#include <metal_stdlib>
using namespace metal;

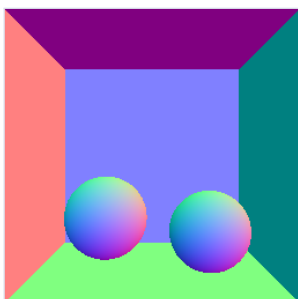
struct GBufferOut {
    half4 albedo [[color(0)]]; // rgb = base color, a = metallic
    half4 normal [[color(1)]]; // rgb = world normal * 0.5 + 0.5, a = roughness
    uint objectId [[color(2)]]; // handy for picking and for the "IDs" debug view
};

struct DebugParams {
    uint view;
    float nearZ;
    float farZ;
}; // 0 = lit, 1 = normals, 2 = albedo, ...

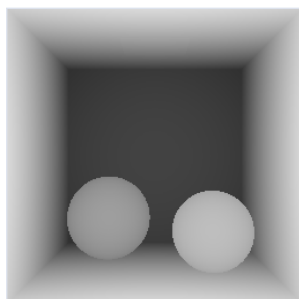
fragment GBufferOut gbuffer_fs(VSOut in [[stage_in]], constant Material& m [[buffer(2)],
    constant uint& objectId [[buffer(3)]] {
    GBufferOut o;
    o.albedo = half4(half3(m.baseColor), half(m.metallic));
    o.normal = half4(half3(normalize(in.normal) * 0.5 + 0.5), half(m.roughness));
    o.objectId = objectId;
    return o;
}

// A full-screen debug resolve: pick what to show with one integer.
float3 debugColor(uint2 px, constant DebugParams& d, texture2d<half> albedo, texture2d<half> normal,
    depth2d<float> depth, texture2d<uint> ids) {
    switch (d.view) {
    case 1:
        return float3(normal.read(px).rgb); // normals
    case 2:
        return float3(albedo.read(px).rgb); // albedo
    case 3: {
        // linear depth
        float z = depth.read(px);
        float lin = d.nearZ * d.farZ / (d.farZ - z * (d.farZ - d.nearZ)); // undo the projection
        return float3(saturate(lin / 30.0));
    }
    default: { // object ids as hash colors
        uint h = ids.read(px).x * 2654435761u;
        return float3((h >> 16 & 255) / 255.0, (h >> 8 & 255) / 255.0, (h & 255) / 255.0);
    }
    }
}
```

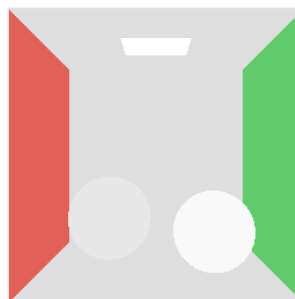
THE FOUR BASIC DEBUG VIEWS (RENDERED BY THE GUIDE'S CPU CODE FOR THE CORNELL BOX)



normals ($\mathbf{n} \cdot \frac{1}{2} + \frac{1}{2}$)



linear depth



albedo



object IDs as hash colors

Debug views are the cheapest way to find bugs. A wrong normal shows up immediately in a normals view (a

flipped sign, an object with inverted winding, a normal map that is not decoded). A depth view shows reversed-Z mistakes and precision problems. An ID view verifies picking and instancing. Build them early and keep them in release builds behind a key.

Chapter 14 in one minute

- A render graph is passes plus declared reads/writes; the compiler culls dead passes, derives barriers and aliases memory.
- Barriers come from reader waits for last writer"; aliasing needs non-overlapping lifetimes of transient resources.
- A G-buffer decouples geometry from lighting; keep it in tile memory when you can.
- Ship debug views (normals, depth, albedo, IDs, heat) from day one.

Debugging and Performance

15

Where to look first when the picture is wrong or slow: validation layers, GPU captures, the debug-view toolkit, and the handful of performance rules that matter on a tile-based GPU. Settings names here are hedged, because they are the kind of detail that changes between OS releases.

15.1 A debugging method for graphics

GPU bugs do not crash with a helpful message; they produce a wrong image. The method that works is to **bisect the pipeline** by making each stage's output visible, from the data up.

BISECT FROM THE DATA TO THE PIXEL: THE FIRST VIEW THAT LOOKS WRONG POINTS AT THE BUG

1. geometry
wireframe, flat color
is it there?

2. transforms
depth and normals
right place and facing?

3. material inputs
albedo, UV, roughness
right data?

4. lighting
one light, one term at a time
right math?

Black screen: stages 1–2. Everything black but geometry visible: lighting inputs (light direction sign, normal facing, sRGB). Weird colors: stage 3 (sRGB flags, UV flip). Noise or flicker: synchronization or an uninitialized value.

15.2 Tools

Debugging toolbox (verify names in the current Apple documentation)	
Tool	Use
Xcode GPU frame capture	Capture one frame; inspect every draw, resource, shader, pipeline state and the dependency graph. First stop for any wrong-image bug
Metal API validation	Checks your API calls (wrong sizes, missing resources). Enabled from the Xcode scheme's Metal diagnostics options (or by environment variable); always on in debug builds
Shader validation	Checks out-of-bounds buffer access and similar inside shaders. Slow: debug only
Metal Performance HUD	On-screen frame time and GPU stats; turned on with an environment variable or in the scheme
Programmatic capture	MTL: :CaptureManager lets you start and stop a capture from code around a specific frame
Debug groups / labels	Name every pass and resource; captures become readable

Hedge: environment variables and Metal 4.1 • I am confident the validation layers, GPU capture and the Performance HUD exist and are enabled from Xcode's scheme settings. The exact environment-variable names and any Metal 4.1 or macOS 27 additions I have not verified here, so check Apple's current documentation or the Xcode scheme editor rather than trusting a remembered spelling.

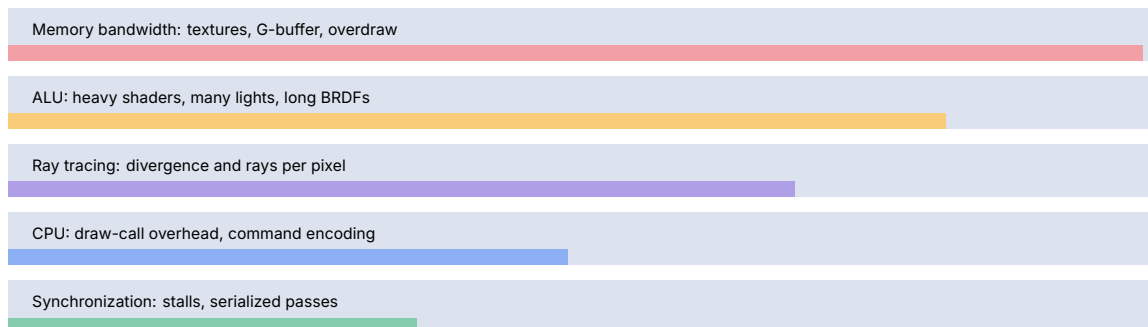
Symptom → likely cause

Symptom	First suspects
Everything invisible	Winding/cull mode, depth clear value (reversed-Z), matrix order, resource not resident
Image upside-down or mirrored	Texture/NDC y flip; a mirror matrix flipped winding
Washed out or too dark	Double or missing sRGB encode; linear versus sRGB texture formats
Black patches / NaN blooms	Division by zero in BRDF (clamp $N \cdot V$, roughness); negative values reaching pow
Flicker between frames	Reusing an allocator or ring slot before the GPU finished; missing barrier
Shadow acne / light leaks	Depth bias; ray origin not offset along the normal
Textures shimmer	No mipmaps, or sampling in non-uniform control flow
GPU hang or garbage	Non-resident resource; wrong argument table slot; struct size mismatch

15.3 Performance on a tile-based GPU

Chapter 1 showed that Apple GPUs shade a screen tile at a time in fast on-chip memory. The rules that follow are about **not wasting that**.

WHAT SPENDS THE FRAME: A TYPICAL ORDER OF THINGS TO CHECK (NOT MEASURED NUMBERS)



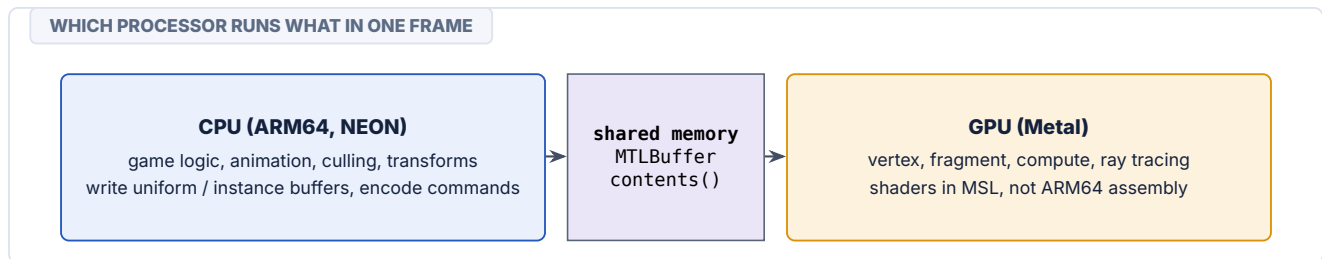
Illustrative ordering only; profile your own frame. In Xcode, the GPU counters and the shader profiler show where time actually goes.

- 1. Measure first.** Capture a frame, read GPU time per pass, and optimize the largest. Never guess.
- 2. Cut memory traffic.** Use compressed texture formats, half-precision where safe, smaller G-buffer formats, and mipmaps. Memory bandwidth is usually the first limit.
- 3. Use tile memory.** Render passes that read what an earlier subpass wrote (G-buffer → lighting) can keep it on-chip. Set store actions to DontCare for attachments you do not need after the pass, and load actions to Clear or DontCare instead of Load.
- 4. Avoid overdraw of expensive shaders.** Depth-sort opaque objects front-to-back; render alpha-tested and blended geometry last (chapter 5's TBDR notes).
- 5. Batch and reuse.** Pre-build pipelines, keep few distinct ones, avoid per-draw allocations (use the upload ring).
- 6. Keep SIMD-groups coherent.** Branch on uniform values; sort or bin rays and materials.
- 7. Prefer half and packed types** in shaders where precision allows; half halves register pressure and bandwidth for color math.

Frame budget • At 60 fps you have 16.7 ms; at 120 fps, 8.3 ms. Ray tracing and denoising eat budget quickly, so decide per effect how many rays per pixel you can afford and what resolution it runs at. Many engines trace at half resolution and upsample.

15.4 Where ARM64 helps: the CPU side of a Metal engine

Shaders do not run on ARM64. They run on the GPU, which has its own instruction set that Apple does not publish as an assembly language you write. But half of every Metal engine is ordinary CPU code, and that code *is* ARM64: updating the scene, building matrices, culling, skinning, filling the uniform and instance buffers, and encoding commands. Knowing how the CPU executes it pays off in a few specific places.

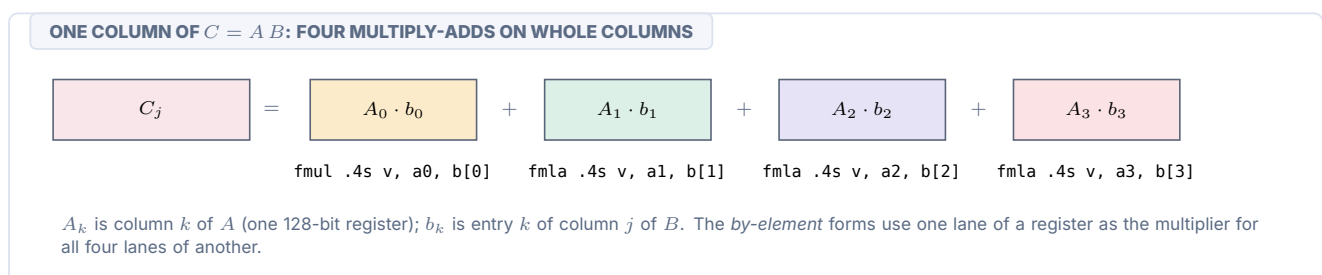


Where it matters

Situation	What ARM64 knowledge buys you
Thousands of transforms or bounding volumes per frame	NEON does four floats per instruction. Write plain loops first; use <code>simd</code> types or NEON intrinsics for the hot ones.
Filling a <code>MTLBuffer</code> through <code>contents()</code>	The CPU writes the very bytes the GPU reads. Lay structs out in the GPU's alignment (chapter 10 explains the <code>float3</code> trap) and write them sequentially.
No-copy buffers (<code>newBufferWithBytesNoCopy</code>)	The memory must be page-aligned; pages are 16 KB on Apple silicon, not 4 KB. Allocate with <code>posix_memalign</code> or <code>mmap</code> accordingly (check Apple's current documentation).
Per-thread encoders and counters	Keep per-thread data on separate cache lines. I believe the line is 128 bytes on Apple silicon (verify with <code>sysctl hw.cachelinesize</code>); 64 bytes would not prevent false sharing there.
Timing CPU work for a frame budget	Read the ARM counter (<code>mach_absolute_time</code> , or <code>__builtin_arm_rsr64("cntvct_el0")</code>) rather than guessing.
A CPU crash or slow function in the engine	Read the assembly: the ARM64 Field Guide shows how to do so with <code>lldb</code> and <code>clang -S</code> .

Example: a 4×4 matrix multiply in NEON

Matrices in Metal and in `simd_float4x4` are *column-major*. Column j of $C = AB$ is a weighted sum of the columns of A , with the weights taken from column j of B . NEON can multiply a whole column by one lane of another vector in a single instruction, which is exactly this shape.



mul: NEON matrix multiply with a portable fallback

✓ arm64 branch compile-checked · logic run with a stand-in NEON header

```

#include <cstring>
#if defined(__ARM_NEON)
#include <arm_neon.h>
#endif

// Column-major 4x4 float matrices (the layout Metal and simd_float4x4 use): m[col][row].
struct Mat4 { float c[4][4]; };
  
```

```
// out = A * B. Column j of the result = A.col0*B[j][0] + A.col1*B[j][1] + A.col2*B[j][2] + A.col3*B[j][3]
Mat4 mul(const Mat4& A, const Mat4& B) {
    Mat4 out;
    #if defined(__ARM_NEON)
        const float32x4_t a0 = vld1q_f32(A.c[0]), a1 = vld1q_f32(A.c[1]);
        const float32x4_t a2 = vld1q_f32(A.c[2]), a3 = vld1q_f32(A.c[3]);
        for (int j = 0; j < 4; ++j) {
            const float32x4_t b = vld1q_f32(B.c[j]);
            float32x4_t r = vmulq_laneq_f32(a0, b, 0); // a0 * b[0] (one fmul by element)
            r = vfmaq_laneq_f32(r, a1, b, 1); // += a1 * b[1] (fmla by element)
            r = vfmaq_laneq_f32(r, a2, b, 2);
            r = vfmaq_laneq_f32(r, a3, b, 3);
            vst1q_f32(out.c[j], r);
        }
    #else
        for (int j = 0; j < 4; ++j)
            for (int i = 0; i < 4; ++i) {
                float s = 0.0f;
                for (int k = 0; k < 4; ++k) s += A.c[k][i] * B.c[j][k];
                out.c[j][i] = s;
            }
    #endif
    return out;
}
```

OUTPUT

```
--- stand-in NEON header (x86 host), the NEON branch:
2000 random products: worst |error| vs double-precision reference = 7.23e-07
translate(1,2,3) * translate(10,20,30) -> (11, 22, 33)
identity * T1 == T1 : yes
--- portable branch:
2000 random products: worst |error| vs double-precision reference = 9.35e-07
translate(1,2,3) * translate(10,20,30) -> (11, 22, 33)
identity * T1 == T1 : yes
```

Here is what clang 18 produced for the Apple target. Four columns times four instructions is sixteen arithmetic instructions, and the loop over j has been fully unrolled and the matrix of A kept in registers ($v0$ – $v3$) for the whole function:

REAL CLANG 18 OUTPUT FOR arm64-apple-macos14 (not run: no ARM CPU here)

```
__Z3mulRK4Mat4S1_
    ldp    q0, q1, [x0]
    ldp    q2, q3, [x0, #32]
    ldp    q4, q5, [x1]
    fmul.4s v6, v0, v4[0]
    fmla.4s v6, v1, v4[1]
    fmla.4s v6, v2, v4[2]
    fmla.4s v6, v3, v4[3]
    fmul.4s v4, v0, v5[0]
    fmla.4s v4, v1, v5[1]
    fmla.4s v4, v2, v5[2]
    fmla.4s v4, v3, v5[3]
    stp    q6, q4, [x8]
    ldp    q4, q5, [x1, #32]
    fmul.4s v6, v0, v4[0]
    fmla.4s v6, v1, v4[1]
    fmla.4s v6, v2, v4[2]
    fmla.4s v6, v3, v4[3]
    fmul.4s v0, v0, v5[0]
    fmla.4s v0, v1, v5[1]
    fmla.4s v0, v2, v5[2]
    fmla.4s v0, v3, v5[3]
    stp    q6, q0, [x8, #32]
    ret
```

What was and was not verified • The ARM64 branch compiled cleanly for arm64-apple-macos14 against clang's real `<arm_neon.h>`, and the assembly above is its real output. To test the *arithmetic* on this x86 machine, I compiled the same function against a small stand-in header that implements the four intrinsics used (with a true fused multiply-add per lane), and compared 2 000 random products with a double-precision reference. The worst error is about 10^{-6} , the same as the portable branch, and

translation matrices compose correctly. This does not demonstrate a speed-up: I did not measure any, and a good compiler may already vectorize the portable version. Measure on your Mac before deciding.

Use the library first • For ordinary matrix and vector maths on Apple platforms, `simd` (`simd_mul`, `simd_float4x4`) is already vectorized and tuned per chip. Reach for hand-written NEON only for batches of work (thousands of items) where a profile shows the CPU is the bottleneck, and keep the portable version as the reference you test against.

15.5 Upgrade checklist for a physically based renderer

Once the basics render, these are the usual next steps, roughly in order of visual payoff per effort:

- Image-based lighting: a prefiltered environment map and a BRDF lookup table for ambient specular.
- Cascaded shadow maps for the sun, or ray-traced shadows (chapter 13).
- Multiple-scattering energy compensation for rough specular (chapter 8's furnace test).
- Temporal anti-aliasing with motion vectors, and temporal denoising for ray-traced effects.
- Bloom and auto-exposure in the HDR chain, then tone mapping.
- Clearcoat, sheen, anisotropy, transmission: extra lobes layered on the base BRDF.

Chapter 15 in one minute

- Bisect: geometry, transforms, material inputs, lighting. Make every stage visible with a debug view.
- Use Xcode's GPU capture and the validation layers; label passes and resources.
- On Apple GPUs bandwidth rules: compress textures, use tile memory, set load/store actions carefully, avoid overdraw.
- Measure before optimizing; budget rays and resolution per effect.
- The CPU half of the engine is ARM64: NEON and `simd` help for batches of transforms, and 16 KB pages and 128-byte cache lines (verify) shape buffer and thread layout.

Reference Pages

One-page lookups to keep open while you code: conventions, the Metal 4 frame recipe, MSL types and attributes, formulas, and a glossary. Everything here is defined and explained earlier in the book.

16.1 Conventions used throughout

The coordinate and storage conventions

Item	Convention (this guide)
Handedness	Right-handed world and view space, camera looks down $-z$
Vectors and matrices	Column vectors, $M\mathbf{v}$, matrices stored column-major; clip = $PV M \mathbf{p}$
Clip / NDC	$x, y \in [-1, 1]$, y up; depth $z \in [0, 1]$ (Metal)
Depth	Reversed-Z: near $\rightarrow 1$, far $\rightarrow 0$; clear to 0; compare Greater; Depth32Float
Winding	Counter-clockwise is front; cull back
Texture / framebuffer origin	Top-left; v grows downward
Color	Do math in linear light; <code>_sRGB</code> formats for color textures and 8-bit targets

16.2 Frame recipe (Metal 4)

One frame, in order

1. Wait on the shared event for the frame that last used this slot; `alloc->reset()`.
2. `nextDrawable; beginCommandBuffer(alloc)`.
3. Reset the upload ring slot. Write per-draw data there.
4. Encode: compute passes (barriers), render passes (barriers), argument table bindings.
5. `endCommandBuffer; queue->wait(drawable); commit`.
6. `signalDrawable; drawable->present(); signalEvent(frame+1)`.

Startup: device, queue, command buffer, allocators $\times 3$, compiler, argument table, residency set (add to queue), shared event, layer.

16.3 MSL quick reference

Types, attributes, address spaces

Item	Meaning
float4x4	Column-major; M * v. 64 bytes
float3 / packed_float3	16 bytes aligned / 12 bytes unaligned. Mind it in shared structs
half, half4	16-bit float: bandwidth and register saving
device / constant	Read-write buffer memory / read-only, small, frequently read
threadgroup / thread	On-chip shared by a threadgroup / per-thread private
[[position]], [[stage_in]]	Clip-space output; interpolated inputs to the fragment shader
[[vertex_id]], [[buffer(n)]]	Vertex index; binding slot in the argument table
[[thread_position_in_grid]]	Global thread ID in a dispatch
[[thread_position_in_threadgroup]]	ID inside the threadgroup
simd_sum, simd_prefix_exclusive_sum	SIMD-group reductions without memory
saturate, mix, clamp, normalize	Clamp to [0, 1]; lerp; clamp; unit vector

16.4 Formulas on one page

THE EQUATIONS THIS BOOK USES MOST

Edge function	$E_{AB}(P) = (B_x - A_x)(P_y - A_y) - (B_y - A_y)(P_x - A_x)$
Barycentric interpolation	$a(P) = w_0a_0 + w_1a_1 + w_2a_2$, with $\sum w_i = 1$
Perspective-correct	interpolate a/w_{clip} and $1/w_{clip}$, then divide
Lambert	$f_r = \frac{\text{albedo}}{\pi}$, light contribution $\propto \max(0, \mathbf{n} \cdot \mathbf{l})$
Cook-Torrance	$f_r = \frac{D F G}{4(\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v})}$
GGX distribution	$D = \frac{\alpha^2}{\pi((\mathbf{n} \cdot \mathbf{h})^2(\alpha^2 - 1) + 1)^2}$, $\alpha = r^2$
Schlick Fresnel	$F = F_0 + (1 - F_0)(1 - \mathbf{v} \cdot \mathbf{h})^5$
F_0 (metalness workflow)	$F_0 = \text{mix}(0.04, \text{base}, \text{metallic})$
sRGB decode	$c \leq 0.04045 : c/12.92, \text{ else } ((c + 0.055)/1.055)^{2.4}$
Monte Carlo	$I \approx \frac{1}{N} \sum f(x_i)/p(x_i)$, error $\propto 1/\sqrt{N}$
Cosine-weighted sample	$r = \sqrt{u_1}$, $\phi = 2\pi u_2$, $z = \sqrt{1 - u_1}$, pdf = $\cos \theta / \pi$
Mip level	$\lambda = \log_2 \rho$, ρ = texels per pixel

16.5 Glossary

Terms

Albedo	Base diffuse color of a surface
BLAS / TLAS	Bottom-/top-level acceleration structure (mesh / instances)
BRDF	Function giving reflected light per incoming and outgoing direction
BVH	Tree of bounding boxes for fast ray queries
Fragment	A candidate pixel produced by the rasterizer
G-buffer	Per-pixel surface data stored for deferred lighting
GI	Global illumination: indirect light from bounces
HDR	Values beyond $[0, 1]$ before tone mapping
Mipmap	Precomputed chain of half-size images
MSAA	Multi-sample anti-aliasing: coverage tested per sub-sample
NDC	Normalized device coordinates after perspective division
PSO	Pipeline state object: compiled shaders plus fixed state
Radiance	Light per area per solid angle along a ray
Residency set	The list of resources the GPU may access
SIMD-group	32 threads executing in lockstep
TBDR	Tile-based deferred rendering
Texel	A pixel of a texture

Where to go next

- Build the triangle (ch. 10), add textures (ch. 6) and the PBR shader (ch. 8), then the render graph (ch. 14).
- Add ray-traced shadows (ch. 13) and a denoiser; use debug views (ch. 14) and captures (ch. 15) constantly.
- Read Apple's Metal 4 documentation and sample code for anything newer than this guide, particularly Metal 4.1 additions.