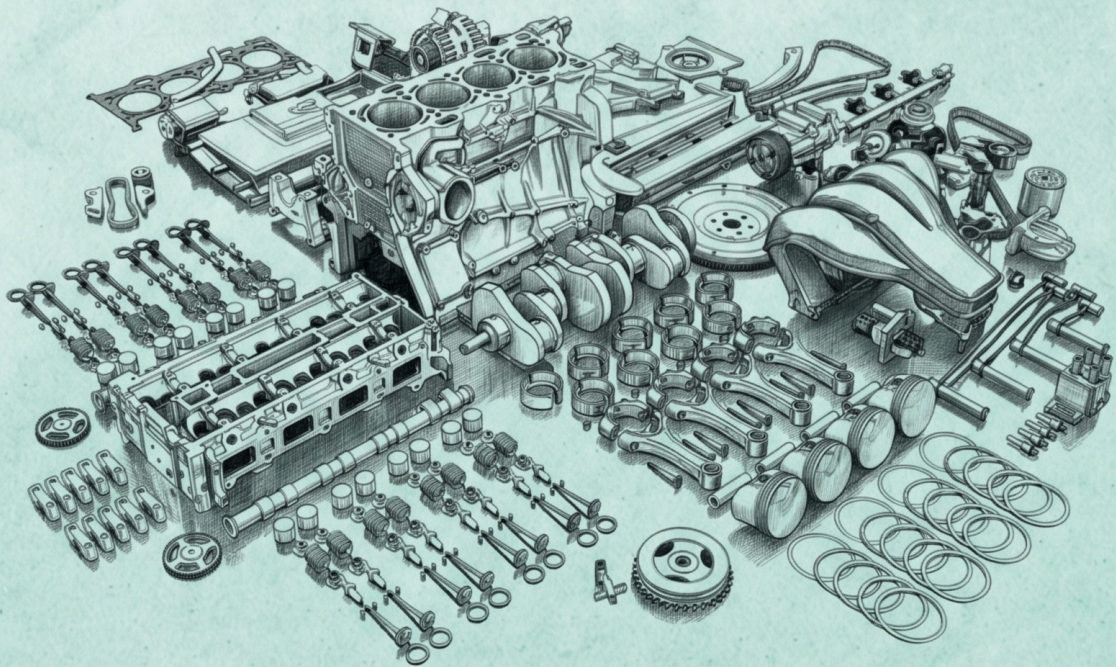


Game++

Game Engine Architecture and Optimization



S. Kushnirenko

Game++

Game Engine Architecture and Optimization

S. Kushnirenko

aka dalerank

Acknowledgments

Every book is a dialogue between the author and its readers, colleagues, and mentors. This book about game development optimization wasn't born out of a drive for academic completeness, but out of an attempt to make sense of years of trial, error, and insight that no one could have lived through alone. This book wouldn't exist without the help of my family, friends, and colleagues from the games industry, and I'd like to thank everyone who helped bring this project to life. First and foremost, my deepest gratitude goes to my family, who taught me the single most important optimization of all: the balance between passion for the craft and time spent with the people you love. To my wife, who understood that solving a technical problem sometimes takes not just time, but the kind of inner quiet you can only find through the support of the people close to you. To my kids, for their patience with our mismatched schedules and for their unconditional love and support despite my being only partly available. Several of my colleagues from 4A Games, Gaijin, Arkane, and Playrix also contributed, either by giving feedback or by helping me shape the structure and content of the chapters. My thanks also go to the engineering teams at these studios for building the incredible game engines whose solutions I partly cover here. I want to thank my mentors (Anton Yudintsev, Alexander Maksimchuk, Sasha Pavlovets, Nikolai Garkavets), who showed me that behind every frame stands a player's experience, and that performance isn't an end in itself but a means to a better, more polished product. Their influence and experience run through every page of this book, even where they aren't named directly. My special thanks go to my colleagues at other game studios: that remarkable community of very different people who are just as willing to share their failures as their successes. In an industry where every project is unique and no universal answers exist, that kind of openness is a rare gift, and many of the ideas in this book are the fruit of a collective search for solutions carried out in countless discussions at conferences, on forums, and over kitchen-table chats. I want to extend my deepest respect to Andrey Karpov, CTO of Program Verification Systems and creator of the PVS-Studio analyzer, for his invaluable work as this book's technical reviewer. His years of experience in static code analysis and his deep understanding of the finer points of C++ proved priceless in checking the technical accuracy of the material.

Thanks to everyone who read the drafts and wasn't afraid to say "this is nonsense, it won't work," that "a pretty theory doesn't hold up in practice," that "an elegant solution creates more problems than it solves." Every skeptical glance and awkward question made the material better, capturing

ideas and solutions that come not from textbooks but from the lived experience of fixing real problems. This book would never have seen the light of day without the publishing professionals Alexey Shevelev, Leonid Kochin, and Oleg Sivchenko, who turned a chaos of technical notes into a coherent narrative. Special thanks to Anatoly Mamaev for editing, for the illustrations in this book, and for proofreading the trickiest parts, and to the readers of Habr for their sharp, spot-on comments on the articles that started this whole project. I also want to wholeheartedly thank the publishing house "BHV" and everyone who helped bring this book into the world, and if it helps even one person avoid some of the mistakes my colleagues and I went through, or sparks a few new questions, then the effort was worth it.

Disclaimer

Optimization in development is always a balance between cost and benefit. Books and articles are full of recommendations, but most of them are either tied to specific architectures or already outdated, and the discussion often gets stuck on low-level details for a particular generation of a particular processor. That kind of advice can be useful in narrow cases, but on real projects aimed at a wide range of hardware platforms, it often turns out useless, so it matters not just to know a bag of specific tricks, but to understand the principles behind performance, how they affect the outcome as a whole, and the "cost" of applying them. In this book I'll try to share my hands-on experience using C++ in the context of game development. It gathers observations, conclusions, and reflections built up over years of work and participation in major game projects at different studios. It's important to understand that this isn't an academic study or an official manual: all of the material is based on personal experience in the games industry, and judgments about what counts as a "good" or "bad" solution were shaped by specific projects, teams, and problems. When I call some solution "successful," I mean it proved itself in practice: maybe it helped solve a hard problem, improved performance, or made the code easier for developers to follow. When I talk about "failed solutions" or "historical mistakes," I mean approaches that caused problems on real projects: made development harder, led to bugs, or made the code harder to maintain. I'll be upfront about the limits of what's covered here, and I reserve the right not to know everything or to simplify complicated topics. C++ is a huge language with a rich history and no shortage of nuance. It's also worth keeping in mind that my view of the language and how to use it was shaped by specific projects and specific problems. In other areas of development (outside games), or on different kinds of projects, my conclusions might not apply well, or might even be the wrong call. Finally, the industry keeps moving, the language keeps evolving, and new tools and practices keep showing up. What looks right today might look outdated tomorrow. My own views might change too, as I pick up more experience. If you decide to keep reading, you do so at your own risk: you might disagree with some of the conclusions, or find some of the ideas questionable or even harmful. That's fine. Game development is too broad and too varied a field for everyone to share the same approach. If you've ever caught an allocation hiding in a hot function, tried to optimize A, or refactored a single-threaded algorithm into a multi-threaded one without redoing the architecture, you'll recognize exactly where the advice and solutions in this book come from.

From the Author...

This book is a collection of thoughts on the C++ programming language, algorithms, and practices in the context of game development: its strengths and weaknesses, practical solutions, and established ways of working. Don't treat this book as a detailed guide to building a game or a game engine from scratch. Think of it instead as "notes in the margins" of the excellent book *Game Engine Architecture* by Jason Gregory. C++ remains the dominant language in the games industry today thanks to its blend of high performance, flexibility, and deep low-level control. But despite its popularity, it carries plenty of architectural and syntactic baggage that's hard to fix, given the need to maintain backward compatibility and stay aligned with where the language is heading. I'm not claiming to have universal answers: in some places I offer concrete ideas and even possible solutions, but more often I focus on pointing out performance problems and discussing general design principles for modern games and game engines. As you read, you'll run into references to material from *Game Engine Architecture*, which is worth reading in full or at least skimming, to follow the gist of these notes. In many places in this book you'll find solutions that diverge from commonly accepted software development practice. But at its core, this isn't an academic study or a how-to manual: it's a personal list of observations, wishes, and grievances about C++ and about approaches to game development, shaped by my own experience. What you're holding is the reflections of a developer who grew up on game engines, late-night debugging sessions, and the struggle to make complex systems run at an acceptable speed.

Wishing you an interesting and informative read! **S. Kushnirenko**
(dalerank)

Foreword

I first got a sense of Sergey's talent for writing when I came across his articles on Habr, where they regularly draw a large readership and strong ratings. What drew me in, too, was his rundown of interesting bugs from the world of C++ game development. The subject of undefined behavior (UB), and bugs in general, is close to my heart and genuinely interesting to me. First, from a practical standpoint: as a source of ideas for new diagnostic rules in PVS-Studio. Second, Sergey really does write in a gripping way, and it's fun to try to guess where his next story about something behaving not quite as intended is going to land. The stories were such a hit that we even put together a fun quiz together, where you have to guess what's wrong with a piece of C++ code. Honestly, the answers are more like educated guesses, since no answer can be truly exact when undefined behavior is involved. But we weren't going for a rigorous test of knowledge anyway. We were going for a mood boost for programmers instead, and on that front, it worked out well. Now you're holding Sergey's new, large-scale work on game development and its optimization. I'm not particularly close to the games field myself, and I won't try to judge, say, the survey of game engines or the specific approaches to boosting performance that are covered in it. But as someone who enjoys "extreme uses of C++," I'd call this book unquestionably worth your attention. You'll find descriptions of plenty of techniques worth reaching for in practice, as well as plenty you shouldn't :) Sergey truly lays bare the full abyss of "the splendor and the horror" that is C++. I wish everyone a pleasant read! And fast, high-quality code, though I know full well that combining the two isn't easy. Good luck :)

Andrey Karpov, PVS-Studio One of the co-founders of the PVS-Studio project, a static code analysis tool.

Who Should Read This Book?

This book is aimed first and foremost at mid-level and senior developers with hands-on experience building C++ applications. If you've worked with commercial engines (Unreal Engine, Unity, CryEngine) or with in-house solutions, and you regularly run into low-level systems problems, designing asynchronous systems, or minimizing dynamic memory use, this book will broaden your understanding of these specific areas. Particular attention is paid to optimization techniques, algorithms, and memory management strategies. The book will also be useful to programmers with no game development background who want to learn the specific patterns and architectural solutions used in the games industry. If you're familiar with modern C++ standards (C++17/20/23), know move semantics and templates, and understand how virtual dispatch and memory layout work, but haven't built real-time systems with hard latency and throughput requirements, this material will introduce you to the particular quirks of game development. It covers approaches to building frame-coherent systems, implementing spatial-partitioning algorithms, design patterns, and profiling hot spots in the context of game systems. If you've ever wondered why a game "stutters" despite seemingly fast logic, how to build a game system with no allocations, why `std::vector` can cause a game to hitch, when it makes sense to give up virtual functions, or how to design the architecture of your game's subsystems so they're easy to profile, debug, and parallelize, then this book is for you. It doesn't just hand out tips on "how to write code": it shows how to tackle architectural and low-level problems that underlie any high-performance system. The material spans a wide range of topics, from low-level language details to the architectural decisions used in game engines and platforms. You'll find practical examples of various C++ features in action, comparisons of different approaches to common game-development problems, a look at the pros and cons of specific language constructs, recommendations for choosing tools and libraries, and reflections on how best practices in the games industry have evolved over time. This book was born as "notes in the margins" of another excellent book that became a bible for several generations of game developers, Jason Gregory's *Game Engine Architecture*. That book is big, thorough, and covers every aspect of building an engine, but it doesn't dwell on the nuances of practical, in-the-trenches development. I understand why it skips the practical examples: including them wouldn't just bloat the page count, it would introduce a lot of subjectivity, and game developers, being creative people, tend to defend their own choices even when those choices aren't always right. The book is

split into five parts. Part one (chapters 0x10 and 0x11) introduces the concepts and architectures of game engines, which will let you follow the ideas and optimizations described in part two. The ideas from part two (chapters 0x20 through 0x25) are then shown in real applications, algorithms, and containers. Part three covers chapter 0x30, which collects standard-library algorithm practices. Part four (chapter 0x40) covers concepts for working with multithreaded logic. And the closing part five (chapters 0x50 and 0x51) discusses game design patterns. Most examples use modern C++11/14/17.

About the Author

I've been programming in C++ for over 20 years now. I started back when the standard library was a lot more modest, and a lot of the language features we take for granted today were just starting to appear. I graduated from ITMO University in St. Petersburg, from the "High-Performance Systems and Networks" department, where I got a solid foundation in systems programming that still comes in handy today with low-level optimization and multithreading. My first steps in programming were on fairly exotic projects: I wrote software for naval training simulators, navigation, security systems, and networking solutions for large enterprises. It was interesting work, but games always had a special place in my heart, so for the last 15 years I've been fully immersed in game development, and I don't regret it one bit. I've gone from simple little games to complex AI systems, animation, and replays, and along the way I've run into pretty much every aspect of game development, from writing my own synchronization primitives to building complex game-logic systems. At Electronic Arts I worked on versions of The Sims and SimCity BuildIt for the low-end devices of the time, which meant inventing creative ways to save on absolutely everything, everywhere. At Gaijin Entertainment, I helped take a team from scratch all the way to the Nintendo Switch and AppleTV stores, shipping ports of Blades of Time and Cuisine Royale, and a bit later worked on the monster AI for the new game in the Metro universe. Right now I'm working on the engine for Age of Empires 2. You'd think it's an old game, but modern performance and feature requirements keep forcing us to rethink architectural decisions, both the old ones from back then and new ones. In my spare time, I'm actively involved in open-source projects. I've written a few UI libraries and created ImSpinner (a collection of animated indicators that unexpectedly became pretty popular on GitHub), and I'm restoring the old 1999 city-builder Pharaoh as part of the Akhenaten project. !!! Everything you read from here on is my personal opinion and doesn't reflect the views of my employer. The examples described in this book were tried out and used on real projects, which you can find online and play yourself: Sims Mobile, SimCity BuildIt, Blades of Time, Cuisine Royale, Deathloop, Metro, Age of Empires 2, as well as various open-source projects. !!!

Table of Contents

Acknowledgments	4
Disclaimer	6
From the Author... ..	7
Foreword	8
Who Should Read This Book?	9
About the Author	11
 Part 0x1 — Foundations: Engines and Architectures	 21
0x10 — C++, Engines, and Architectures	22
Why C++?	23
What Is a Game Engine?	24
The First Big Question: Why?	25
The Second Big Question: Do You Even Need It?	28
Libs + CMake != an Engine	29
Handy Essentials	30
Programming	31
The Main Window	31
User Input	33
Graphics	35
Resources	36
Audio Output	37
Physics	38
AI	39
Scripts and Configs	43
Networking	46
UI	49
Tools	50
Architecture	51
Game-Engine	54
0x11 — Game Engine Architectures	55
0x12 Unity (Unitary Architecture)	58
0x13 Big Ball of Mud	59
0x14 Unreal Engine (Layered Architecture)	63
0x15 CryEngine (Microkernel Architecture)	66

0x16 Dagor (Data-driven Architecture)	69
0x17 X-Ray Engine (Monolith Architecture)	72
0x18 Godot (Modules/Services Architecture)	75

Part 0x2 — Low-Level Optimizations 78

0x20 — Optimizations 79

Types of Optimization 80

Architecture-Level Optimizations (~50% of the gain)	82
Algorithm/Data-Structure-Level Optimizations (~30%)	83
Source-Code-Level Optimizations (~10%)	84
Game Replays (~time?)	90
Component Isolation (~5%)	91
Specific Low-Level Optimizations (<3%)	91
Everything Else (~10%)	92
Benchmarks	92
Measurement	93
Changes and Engineering Intuition	94
Hot Functions and Bottlenecks	95
Hot Functions (Hotspots)	96
Bottlenecks	97
Misconceptions and Myths	98
Tools	100
Theory and Terminology	102
The Story of One Optimization	105
Packing Booleans into Bits	110
Conclusions	119

0x21 — String Optimizations 121

0x22 String Interning (String Pool) 122

Not Quite Strings	125
Implementing xstring	129
How to Use It	132
Impact on Architecture	133

0x23 Identifiers 135

0x24 Short String Optimization (SSO) 137

0x25 Short-Lived Strings (Short Live String) 139

0x26 Long-Lived Strings (Long Live String) 143

0x27 String Arenas 144

0x28 — Array Optimizations 147

ox29 Optimizing Arrays (Cooking Vectors)	148
Vector Downsides	152
Fixed Array	159
Stack Size	163
Static Vector	164
Hybrid Vector	169
Hot/Cold Data Layout	171
ox2A — Allocator Optimizations	174
All These Different Allocators (Dancing with Allocators)	175
Memory Management	176
Manual Memory Management	178
Automatic Reference Counting (ARC)	181
Unique Ownership	183
Fragmentation	184
Internal Fragmentation	185
External Fragmentation	185
Block, Header, Lifetime	186
RAII (Resource Acquisition Is Initialization)	187
Memory Leak	188
Double Free	189
Dangling Pointers	190
Heap Corruption	191
Defensive Programming	192
Smart Pointers	193
Custom Allocators	194
Lazy Destroy	194
Memory Arena	195
Object Handles	196
Dependencies	197
Cascade Cleanup	200
Data Race	202
Memory-Safe Algorithms	203
CPU Registers	204
Interior Mutability	205
Memory Mapping	205
DRAM (Dynamic Random Access Memory)	206
DMA (Direct Memory Access)	207
Alignment	207
Instruction Lifecycle	208
Prefetching	209

Stack vs. Heap	211
Linear Allocator	213
Step-Back Allocator	216
Frame Allocator	219
Double-Frame Allocator	221
N-Frame Allocator	225
Stack Allocator	228
Double-Stack Allocator	234
Pointed Double-Stack Allocator	238
Triple-Stack Allocator	240
Pool Allocator	243
Free List Allocator	248
Segregated Free Lists	253
Slab Allocator	256
Buddy Allocator	258
Region-Based Allocator	260
Thread-Cache Allocator	268
Fibonacci Allocator	272
Compacting Allocator	276
Two-Level Segregated Fit (TLSF)	279
"Color-Based" Allocator	283
TTL Allocator (Time-to-Live, Tracked-Time-Live)	285
Chaos Allocator (Aggressive Address Randomization)	286
Allocators != a "Silver Bullet"	287
ox2D — Container Optimizations	288
Containers	289
Iterator Invalidation	290
Fixed Array	290
Ragged Matrix (Jagged Array)	293
Dynamic Matrix	295
Dynamic Array (Vector)	297
Stable Vector	299
Sparse Array/Vector	304
The "lazy sorting" paradox	305
Packed Vector	305
Hybrid Vector (Small Vector)	307

Dirty Vector (Deferred-Deletion Array)	312
Buddy Array (Interleaved Array)	316
Double-Ended Queue (Deque)	318
Block-Compressed Deque	320
Bounded Priority Deque	322
Intrusive Deque/List	324
Expandable Vector	328
Log-Structured Merge Tree (LSM)	329
Octree	330
Quadtree (Quadtree/btree)	334
Circular Buffer	336
Memoization Hash Table	339
Handle Array	343
Texture Atlas	347
Slot Array	352
0x2E — Heap of Fun	354
Heap of Fun	355
The Stack	362
Dynamic Memory (The Heap)	367
Static Memory	374
The Trade-offs of Dynamic Objects	376
Stack Determinism	377
Leaks and Optimizations	378
Just Don't Use new?	379
Local Strings	381
Local Polymorphism (Ad Hoc Polymorphism)	383
Optimizing Vectors	389
Optimizing Map	393
Optimizing a Tile Map	396
Working With Preallocated Memory (Preallocations)	398
Expensive Logging	401
Part 0x3 — The Standard Library	406
0x30 — STL and Algorithms	407
0x31 Nonstandard Standards	408
Using SIMD	412
0x32 Algorithms	413
The Orthogonality of Data and Algorithms	417
Iterators	420

Shader Caching (A Practical Example)	424
How Do You Do It With Algorithms?	426
Commonly Used Algorithms	428
Part 0x4 — Concurrency	475
0x40 — Multithreaded Programming	476
0x41 Concurrency	477
Scalability	478
Contention	479
Separate State	480
Shared State	480
Shadow State	481
0x42 Working with Threads	482
Real-World Analogy	483
Thread Problems	484
0x43 The Story of a Bug, or the Multithreaded AI That Beat Everyone	488
Non-Blocking Logic	488
How Was It Before C++11?	490
Non-Blocking Algorithms (non-blocking, lock-free, wait- free)	494
Why Move Away from Mutexes at All?	496
Atomic Operations	498
Spinlock	499
Thread Starvation	500
Algorithmic Starvation	501
Cache Effects	502
A Good Spinlock for a Game	522
Practical Tips	524
0x44 Tasks	527
Hitching, Stutters	530
Reaching into the Resource System (I/O Hitching)	531
Mishandling Threads (Thread Hitching)	531
Garbage Collection (GC Hitching)	532
Hardware Problems (Graphics Hitching/Stutters)	532
0x45 Worker Threads (Workers)	533
0x46 The "One Thread per Core" Scheme	534
0x47 The "Two Threads per Core" Scheme	535

The Independent-Tasks Model (Embarrassingly Parallel Model) ..	536
The Data-Sampling Model (Stateless Workers)	538
Task Chains (Workers Chain)	539

Part 0x5 — Design Patterns 547

0x50 — Design Patterns in Actions 548

0x51 Frameworks Die, Math Doesn't Age 549

The Blue Book	551
Felt-Pen Syndrome	553
Pattern Levels	554

0x52 Command 556

Downsides	559
Room to Extend	559
Compatibility	560
Bad Implementation Examples	560

0x53 State 563

Finite State Machine	564
Deterministic (basic) FSM	566
Nondeterministic FSM	567
Pushdown-State FSM (FSM + pushdown states)	568
State-based/Object-model FSM	569
Room to Extend	572
Influence on Architecture	572
Downsides	573
Bad Usage Examples	574

0x54 Shared Object/Flyweight 577

Influence on Architecture	578
Downsides	579
Compatibility	580
Bad Usage Examples	580

0x55 Observer 582

Room to Extend	587
Influence on Architecture	587
Downsides	588
Compatibility	589
Examples of Bad Usage	589

0x56 Event Bus 592

Room to Extend	593
----------------------	-----

Influence on Architecture	594
Downsides	594
0x57 Dirty Flag	596
Influence on Architecture	599
Downsides	599
Compatibility	600
Examples of Bad Usage	600
0x58 Update Layer	603
Close Layer	604
Mid Layer	604
Far Layer	605
Frozen Layer	605
Implementation	606
Influence on Architecture	607
Downsides	608
Compatibility	609
Examples of Bad Usage	609
 0xFF — Games Are Art...	 612

0X1

Foundations: Engines and Architectures

0X10

C++, Engines, and Architectures

Why c++?

C++ remains the dominant language for game and game-engine development, despite how archaic and complicated it can look from the outside. The reason comes down to a unique balance between low-level control and high-level expressiveness that's critical in the games industry. On one hand, C++ gives you full control over system resources. Developers can manage every byte of memory, tune processor cache (throughout the text, the author's own spelling of this term is kept as written) usage, and build synchronization and parallelization systems around a specific architecture. You can write code that runs efficiently on "weak" hardware, last-gen consoles, or mobile devices. You can manually control memory and data layout to squeeze out maximum cache efficiency, which often ends up being the deciding factor in the fight for performance. On the other hand, modern C++ gives you powerful high-level abstractions for building complex systems. Classes, inheritance, templates, containers, smart pointers, RAII, lambdas: this whole arsenal lets you build large-scale architectures with modular logic, entity systems, and event mechanisms, and when used sensibly, without going overboard on overloading and metaprogramming, the code stays expressive and maintainable. Alternative high-level languages (Python, JavaScript, C#, or Java) have undeniable advantages, like automatic memory management, convenient syntactic sugar, a rich standard library, and dramatically better readability. That's exactly why they're popular with designers, technical artists, and UI programmers for scripting tasks in games. But these languages have critical downsides for building game engines. Dependence on a virtual machine or runtime, the unpredictable behavior of a garbage collector that can freeze execution at the worst possible moment, the inability to precisely control operation timing, dependence on external runtimes, and suboptimal memory layout schemes: all of this makes them unusable for a game engine's core subsystems. In game development, especially at the engine level, limitations like these become unacceptable. A garbage collector, extra megabytes for a virtual machine, or inefficient use of the processor cache end up causing hitches and frame-rate drops at random moments.

Physics simulations, rendering systems, network code, and the engine's core logic all demand predictable performance and maximum resource efficiency, and they don't forgive mistakes. C++ stays relevant in the games industry not because it's perfect, but because every alternative is too abstract to solve the real technical problems of game systems. It's a conscious trade-off: the language is complex and potentially dangerous, but when used

correctly it delivers the efficiency that modern games need. As long as performance requirements and resource control stay critically important, C++ will hold its ground in game development.

What Is a Game Engine?

Back when the trees were tall and game engines were small, there wasn't much of a choice about whether to write your own: if you didn't have an engine, you basically didn't have a game. Some people bought someone else's engine and enjoyed the lovely "reinvented wheels" of an imported build, while others built their own, which took months if not years. To understand how (and more importantly, why) the mechanisms inside a given game or engine are used the way they are, to be able to spot, track down, and fix performance and architecture problems, you have to understand how game engines work and how they came to be. John Carmack said the best way [to make games] is to build your own engine ("The right move is to build your own engine"), to which plenty of people would object that it's really not that simple. But Doom's "dad" isn't just known for his contributions to game engine development; he's also well known for being pretty vocal and critical about the state of game engine development in general, and about the advantages of building your own tech instead of using something off the shelf. This philosophy, which he's held onto throughout his career, comes from the idea that your own engine gives developers full control over the technology and the ability to build solutions tailored to their project's specific needs. But you know, everyone's a little bit right here: there's no universal answer, it all depends on the specific situation. Maybe building your own engine isn't actually as hard as it looks? Why even bother? And what does "building a game engine" even mean? I've worked with a bunch of different game engines, built a couple of my own from scratch, and two more before those ended up in the trash. On the first one (well, third, really), three commercial games shipped, though without much success, and the engine eventually got sold to a studio in St. Petersburg for "a wild stack of evergreen raccoons" 15 years ago. I've watched developers use other general-purpose engines, complain about their problems, and talk about their own homegrown solutions. I've studied, and keep studying, the source code of other engines (open and otherwise), though I'll admit my experience might be a bit biased or one-sided: I always wanted to make strategy games, but somehow I kept getting offered shooters (shooter: a video game genre centered on shooting) instead.

The First Big Question: Why?

Deciding to write a game or a game engine can feel like madness. Or like a brilliant idea. In reality, it's both. This is one of those cases where the journey is a lot more interesting than the destination.

...first, second, and third: it's a ton of fun, I promise

Building an engine is like LEGO, but for adults (and usually solo). You get to bolt on rendering, a particle system, hot script reloading: you'll laugh, you'll be surprised, maybe you'll even curse a little, but it definitely won't be boring.

...fourth: it's experience that companies pay good money for

Precisely because you went through all those iterations and rewrites, you came out with something you can't buy from any textbook: a real understanding of how a game engine works under the hood. That hands-on experience is your main asset on the job market, and studios are willing to pay well for people who understand the inner workings of game tech: most people know how to use ready-made solutions, but when a problem shows up at the engine level, you need someone who's "been in the trenches" and knows where to look for the root cause.

...you will absolutely learn a ton along the way

It'll help you understand other engines better, and not just engines. You'll develop real engineering instincts: an understanding of what "heavy algorithms" actually means, why it's a bad idea to load textures mid-frame, and what cache-friendly data structures, designed to make the most of the processor's cache, actually look like. None of this comes from books: it comes from firsthand experience. And once you eventually give up on writing your own engine (which, let's be honest, you probably will), you'll start writing games and software in a completely different way.

...speaking of the trash can

Your first engine is going to end up there, and in my experience, so will the second. That's normal, and it's part of the learning process. There's a concept called the "*rule of three systems*": the first is the trial run, the second is the attempt to do everything "the right way," and the third is the one that actually starts working. So go ahead, write the code, delete the solutions that don't work, and start over. Every "failed" version isn't wasted time: it's priceless experience that brings you closer to that third system, the one that actually works.

...your own engine is free, exactly as free as your evenings are

As you can see, it's not free at all. It's hundreds of hours spent evenings and nights. It's arguing with yourself. It's refactoring the thing you wrote full of inspiration two weeks ago. It's "it mostly works" and "I'll finish the UI

later," but in exchange, you get understanding and confidence in your own decisions.

...you can, and should, rework everything to fit your own wants and needs

Want hot-reloading configs in JavaScript? You'll have to dig into JavaScript, and even a bit into its VM. Tired of the outdated way of loading textures? Build your own runtime atlas straight from files on disk. There are no limits here, other than your own skills and time. Weird desires are a good thing. They're what leads to unique solutions.

...full control over how everything works on the inside

Which systems are involved. How the world update runs. What data structure holds the game objects. There's no "black box" for you: just a call stack and the urge to simplify everything.

...you get to use your favorite programming language.

Love Rust? Use it. Love C++? Welcome to pain club. Some people write engines in Go. I've seen engines in Lua, in Python, in Haskell. Even in Erlang. I haven't run into one in COBOL, but I believe one exists out there somewhere. What matters most is knowing why you picked that particular language.

...you can ship a tiny distributable

Take my hobby project, for example: its binary comes in at around 2 MB. It has scripts, configs, and even some assets packed right in. Everything else gets pulled from disk. For a small game, that's an absolute luxury, a dream, especially compared to engines where a bare "Hello World" weighs 200 MB.

...you can open-source your engine

And that's a great idea. First, you get to see how other people use it. Second, you find the people who actually managed to build it. There won't be many: something like 0.1% of everyone who downloaded it. But those are exactly the people worth inviting onto your team. Epic Games, by the way, did exactly that: active Unreal contributors ended up becoming official employees.

...and you can even sell it

But be careful: supporting an engine to satisfy other people's wishes is its own special circle of hell. People will message you with things like "can you make it work like Unity, but just a little different." And that road never ends.

...you'll never be disappointed that the engine developers didn't implement the feature you needed

Because here, you're the developer. Want a new navigation system? Go build it. Need to extend the save system? Go for it. And there are no bug reports where someone tells you, "Planned for Q5 of next year."

...and finally: technical interviews

When you have your own engine, the conversation almost always turns interesting. You're no longer just answering questions: you get to ask them yourself (and speak with authority): "So how do you handle updating game objects? How's asset loading implemented?" People start listening, because they know they're not talking to just a programmer: they're talking to an engineer who built their own solution and isn't afraid to show that knowledge off.

For me, the first two points and the last one mattered the most. Learning is fun; learning while building a game is fun squared. And honestly, I just love programming: doing the things the language "supposedly" doesn't let you do, digging into algorithms and systems, poking around in the fiddly details and quirks of the language. The heart-to-heart chats at interviews are just a nice bonus on top. Game developers always complain about engines, no matter how slick and convenient they are: this doesn't work, that broke, the new thing doesn't support the old thing, and that old thing was requested ten years ago and still hasn't been added. Of course, there are reasons behind all of it: if you've ever worked at a big company, you know exactly what I mean. But it's always easier to blame someone else than yourself. If you need that feature so badly, why not just go write it yourself? Well, because it's not your engine. The engine belongs to someone else, but the frustration is all yours. That's expected. Just as it's expected that in your own engine, the feature you need actually gets added the moment you truly need it. By the early 2000s, the term "game engine" was already firmly part of a developer's vocabulary, but most projects were still, as they say, "homegrown," and the vast majority of the tech was written from scratch: from a custom renderer to a homemade animation system and hand-rolled physics. Back then it was completely normal to build an engine for one specific game, tailored to its genre, platform, and technical constraints. Since then, the industry has changed a lot: games became a mass-market, commercially mature product, and development technology became more standardized and modular. Modern game engines (whether Unreal Engine, Unity, CryEngine, Source, or a big studio's in-house tools) have turned into full-fledged SDKs that let you build not just one project, but an entire lineup of games, often across different genres, on a single technology base. Even though specific engine implementations can vary wildly, by the mid-2000s a more or less stable set of core subsystems had taken shape: graphics, physics, animation, audio, AI, UI, core system logic, replication and networking, in-game scripting, data systems, and so on. What's more, even within these components, certain patterns and architectural approaches started showing up again and again from project to project: things like ECS (Entity-Component System), behavior trees (BT), deferred rendering, or a rollback-capable synchronization system. But information on each of these topics had

to be pieced together bit by bit: an article here, a GDC talk there, a chunk of code from GitHub somewhere else. There were books on graphics, on physics, on AI architecture, but it was all scattered knowledge. There were practically no books that tied everything together: that explained not just "what this is" but "how all of this lives together under one roof," how to design a game's architecture while accounting for technical constraints and the realities of production, the way GEA does.

The Second Big Question: Do You Even Need It?

Doing your own project is always hard and takes a lot of time: like, genuinely a lot of time. Of course, a lot depends on your end goals, but you shouldn't expect to knock out an Unreal Engine clone in a year, especially given that engine has been built for about twenty years now by a team of top engineers, and these days the community pitches in too. Your own engine will always be missing something. You'll need to put in years of hard work just to reach a fraction of the functionality and usability that mature engines already have. You will never be able to compete with big industrial-grade engines like Unreal, Unity, CryEngine, or even Godot. Unless, of course, you're a large, well-funded studio, but in that case you probably don't need this book much anyway, since you already have your own engine, or you're using one of the ones listed above. Here's something you need to understand: the cost of developing your own engine will never pay for itself: there's no moment where, as of yesterday, the engine suddenly started turning a profit. No, even at the top studios, it's a cost center. Games make money, not engines... According to Unreal's own figures, engine development costs over just the last four years topped \$160 million. Epic is effectively shipping a new AAA game every year, we just never see it. In 99 cases out of 100, if building the engine itself is your actual goal, you'll never even get to the game part. That's a direct path to the project getting scrapped. Most people are going to think you're not very smart, simply for spending your time building your own engine.

Tip

If you want to ship a game as fast as possible, just pick an existing engine and use the functionality it already has. But if you want to grow as a developer, build yourself a foundation for future projects (not just in game-making), and you've got the time, energy, and patience, then go for it: your own engine is probably the best use of that knowledge you'll find! On top of that, once you've written a solid foundation, you can reuse it again and again, making it better with every new game.

Libs + CMake != an Engine

You've read this far, which means I haven't managed to talk you out of this thankless pursuit, so let's keep going... Are SDL/SFML/Allegro game engines, even though they're behind countless indie games? Definitely not: they're libraries that wrap platform-specific code, even though they let you do a lot of what a basic game engine can do. What about Vulkan/DX12? Also no: those are graphics APIs. FMOD/WWISE? Also no, though they let you build a 2D/3D scene, just a sound one. So, the conclusion is:

Libs + API != an engine

Now picture a small project about exterminating spiders and building a base on another planet, one that combines Allegro, DX11, and stb (a grab bag of header-only solutions for every occasion) just to draw a few moving sprites on screen. You probably recognized Factorio in that description: that's exactly how it started, as an Allegro demo with a couple of sprites:

Allegro + API + stb + talent = Factorio

Is that a game engine? Still no, though it's getting close: a game engine only really becomes one once there's a game (or games) made with it. Until there's a game, it's just a pile of libraries glued together somehow with CMake. Maybe it's just a combination of graphics, animation, and input handling? So here's where that leaves us:

Factorio != engine + Factorio = a game

A game engine without a game is just a bag of tools, libraries, and other components meant to help make games. A game without an engine is still a game. If it feels like that definition of "an engine without a game" is vague and pretty much useless, you're not imagining it. Do you really need elaborate material editors and shaders a mile long, seamless geometry with global illumination to go with it, 3D physics with inverse kinematics, and network multiplayer support, if you're making a 2D pixel-art platformer? Unreal Engine has all of that, and at a pretty solid quality, but do you actually need all of it? It takes experience, getting used to, and knowing how to configure and use it. All a platformer really needs is handling input, rendering sprites, maybe even just slapping raw PNGs on the screen, and playing sound. All of that can be written in a couple of days from open YouTube tutorials. And those evenings will have been well spent, instead of wrestling with a character controller inside some editor. And now you've finally gotten to the whole point of this exercise: actually making the game. But what if you want to make an AAA hit with top-tier graphics, physics, and everything else? Then yes, you're going to need all those hugely complex systems. And a couple of decades of free time, or a solid team of sharp engineers working on nothing but the engine every single day to pull it off. Yeah, modern game engines have come an impossibly long way from a retro

platformer. Even so, most game engines have an architecture that pulls all these subsystems together into one whole. Not necessarily all of them right away, but it's worth understanding that your particular engine can contain only part of that list and still count as a game engine. Not every game needs every feature at once: if it did, no indie game would ever have shipped. An engine can be small, built specifically for your game. But let's be honest, we've been spoiled by what big engines can do, and we carry those expectations over, expecting the same behavior from everything else.

Handy Essentials

But if an engine is built from the start for one specific game, having only a handful of features is also perfectly fine. When you make games (on your own engine, on someone else's, from scratch, on the wreckage of a previous project), you start noticing recurring patterns, and everything that looked like a "temporary solution" at first eventually either turns into a load-bearing foundation or gets thrown in the trash. That's how the "standard toolkit" of must-have components comes together. First up: visual level editors and UI tools: you shouldn't be hand-writing levels, even for a one-evening prototype. Once you've got something moving on screen, the next question inevitably shows up: updating the logic. The naive approach is to cram everything into one update function (`Game::Update()`), and whoever gets updated first, gets updated first. Then update priority shows up, then dependencies between systems, and then the question comes up: if physics runs separately, what happens to the logic that depends on collisions? Even in a small game, it's better to design your logic as an independent system rather than a chain of hacks that depend on the order of elements in an array. It's better when components know absolutely nothing about each other: they subscribe to events, and they don't care who's calling them. Next comes the problem everyone eventually runs into: the scene/level system. It seems obvious at first: you've got a menu, a game, a cutscene. But once a level loads inside another level, and one game mode switches to another without going back through the menu, and you need to "freeze" a scene, show a tutorial overlay on top, and then go back, that's when the scene system starts falling apart, unless it was designed for this from the start. A scene isn't just a list of objects: it's a managed container for logic, data, dependencies, and transitions. And since we're talking about objects, we can't skip the entity manager, a thing that seems pointless and redundant at first, and then turns out to be the center of the whole architecture. It's responsible for creation, destruction, identification, tagging, component access, and sometimes debugging too. And speaking of saving: the game-state serialization system. A lot of people think this is only needed for save files, but it's actually needed for everything: autosaves, passing state between scenes, saving

configuration, replays, rollbacks: you'd be amazed how many problems disappear once you can just "pack everything into a box and restore it later." And finally, the variable inspector. I don't mean a full-blown debugger: I mean a panel where you can watch variable values, ticks, and flags in real time. Where you can hit "pause," check what a particular NPC is doing, change its behavior, flip on some debug info, and you'll be amazed how many bugs just vanish, simply because you can finally see what's actually happening inside the game's logic.

Programming

It might sound strange, but classical programming isn't actually required for implementing game logic. Just look at visual-programming systems (Blueprints, VisualScript, ICS), which have existed for decades and let you build complex mechanics without writing a single line of code in the traditional sense. This is especially popular among designers and artists who want to quickly prototype ideas without diving into programming-language syntax. Designers write code too: visual programming is a fully legitimate form of programming, it's just that words like class, if, and for look different. But the point here isn't about terminology, it's about the practical side of things. Implementing visual scripting in an engine requires building a whole set of specialized tools: node editors, a visual debugger, data-flow visualization systems, optimizers, and a lot more. That makes the engine itself a lot harder to build: for instance, behind Unreal Engine's Blueprints sits a massive amount of low-level C++ code, comparable in scale to the renderer or the animation engine. Classic code remains the most universal and flexible way to implement logic, but it keeps losing ground. It isn't limited by the constraints of visual tools, and it lets you work directly at the level of system resources. But the classical approach drags along well-known problems: recompiling, stopping the engine to rebuild, and losing time on all of that, not to mention that teaching designers a language like C++ costs a lot in both money and time, and the price of a mistake is the same as for a game programmer: a crashed engine (crash: the game "crashed," meaning it suddenly closed due to an error, often without saving progress, which almost always means more development time lost).

The Main Window

Most games need a window to run in at all. Browser games aren't an exception here either: the "window" might be the entire web page, or it might be a specific element that the game runs inside of. Creating a window is usually done through platform-specific mechanisms, and these low-level interfaces tend to look clunky and awkward, and on top of that, for every

platform (Windows, Linux, macOS, iOS, Android) you'll need to write separate window-creation code. But that code only gets written once, and nobody ever comes back to touch it again. Besides the window, there's usually a graphics API context (OpenGL, Vulkan, DirectX), which also gets created through some scary platform-specific calls. But again, you only have to write that once, or you can just grab a ready-made library that handles all of it for you (SDL2, GLFW, SFML, or any of a thousand other libraries). Once the window is created, the next step is setting up the game loop: the thing that will handle events, update the game logic, and redraw the screen. **The Game Loop** Here's what practically every game and game engine has in common, even though the loop itself might be buried pretty deep inside the engine, replaced with abstractions, or turned into callback functions. But at the end of the day, the game loop's code looks something like this:

```
int main() {
    // Create the window and its associated system
    resources
    createWindow();
    // Initialize subsystems (graphics, audio, game state,
    etc.)
    init();
    // The main loop runs until the application decides to
    quit
    while (isRunning()) {
        // Handle events (keyboard, mouse, etc.)
        handle(inputEvent());
        // Update game/application logic (physics, AI,
        timers)
        updateFrame();
        // Draw the current frame to the window
        drawFrame();
    }
    // Tear down the game state and free resources
    shutdown();
    // Destroy the window and remove platform handlers
    destroyWindow();
}
```

The game loop is what makes the game actually run: without it, main would just finish immediately and the game would close. There are plenty of ways to expose this loop in an engine's API. It can be fully explicit, where the engine's user writes `while(engine::isRunning())` somewhere in their own

main function. Or it can be hidden inside the engine, say, in some `engine::run()` function, which in turn calls a set of user-provided callbacks. Whatever you choose, you're still going to need a game loop.

User Input

This is one of the basic building blocks of any game, even if all the player does is "tap the hamster." At the most basic level, all input is handled through platform-specific APIs like WinAPI, X11, or Cocoa, but working with them directly isn't fun: they're not complicated, just extremely verbose, and they require separate code for every platform you support. But almost all of these APIs let you funnel events into something like a `Game::pollEvents()` function, which you can then use to pull events from the user one at a time, in order. When it comes time to wire input into a game engine, there's more than one way to do it. One option is to give the user direct access to the event system. In this case, the game developer calls `pollEvent()` themselves and decides how to handle whatever comes back. The basic structure usually looks like this: on every iteration of the game loop, an event-handling function gets called, which walks through every event that came in, one by one. Depending on the event type, the appropriate functions get called or specific actions get taken: handling mouse movement, a key press, or the window closing, for example. This approach gives you maximum flexibility, but it also demands a lot of code from the user and a solid understanding of the event structure. In practice, game developers almost always rely on cross-platform libraries or the built-in input systems of engines like Unity, Unreal Engine, or Godot. These solutions take on all the "dirty work" of talking to the operating system and expose one clean, unified interface for handling keyboard, mouse, gamepad, and touch input. Here's an example:

```
while (engine.isRunning()) {
    Event event;
    while (engine.pollEvent(event)) {
        switch (event.type) {
            case Event::MouseMove:
                application.onMouse(event.mouse); break;
            case Event::KeyPressed:
                application.onKey(event.keyCode); break;
            case Event::WindowClose: engine.quit(); break;
        }
    }
    engine.update(); // Game logic
    engine.render(); // Render the frame
}
```

Another option is to build event handling into the engine itself and only expose handler-function calls to the user. That makes initial development simpler and faster, but it requires a well-thought-out architecture for registering those handler functions. This approach makes life a lot easier for the game developer, especially in the early stages of a project. Instead of having to poll input devices and deal with complex event logic themselves, the programmer just registers handlers for the events they care about: key presses, mouse movement, touch events. The matching handlers get called at the right moment, with all the necessary event information passed in, convenient for simple games and prototypes, where development speed matters and complex interaction logic isn't needed yet. Popular libraries (frameworks) tend to implement exactly this kind of event model, but as a project grows more complex, this architecture can become a source of problems. The main difficulty is building a flexible system for registering and managing event handlers: you need to support adding and removing callbacks at runtime, setting priorities, and filtering events by game context, and with a large number of registered handlers, a single event can trigger a chain of calls that nobody planned for back at the start of development.

```

void onMouseMove(MouseEvent event) {
    application.onMouseMove(event.mouse);
}
void onKeyPressed(KbEvent event) {
    application.onKeyPressed(event.key);
}
void onWindowClose(AppCloseEvent) {
    engine.quit();
}
int main() {
    engine.subscribe(onMouseMove);
    engine.subscribe(onKeyPressed);
    engine.subscribe(onWindowClose);

    while (engine.isRunning()) {
        updateFrame();
        engine.update(); // Game logic
        engine.render(); // Render the frame
    }
}

```

Graphics

We've already got a window. We can even sort of react to the mouse and keyboard. But... we still can't show that reaction on screen. Time to tackle graphics. Honestly, 3D graphics always gave me a hard time, which left me with a healthy dose of envy toward my colleagues who could whip up renderers for anything, from consoles to phones, without breaking a sweat. Most window-creation libraries, like SDL, already come with 2D graphics support: you can draw straight into the window, use textures, or manually fill pixels with whatever colors you want. Primitive, but honest. Say your engine is built around 2D games with tiles and sprites. Then you'll probably need to implement something like `drawSprite(image, x, y)`, maybe with support for various nice-to-haves: scaling, rotation, color tinting, desaturation, alpha blending. Under the hood, this can be done by hand: copying pixels directly with some math, or with no transformation at all (blitting). Or it can go through another library like Cairo, which turns out to be a surprisingly capable renderer under the surface, able to do if not everything, then a whole lot. Or through OpenGL/DX/Vulkan, if your soul is craving some hardcore speed. Or you can skip specializing entirely and just hand the engine's user a generic API, the way most SDL/SFML/Allegro libraries do: let them figure it out themselves. In my implementation for "Pharaoh," there's a wrapper around SDL (which can tap into native graphics

APIs), but there's no full-blown renderer in the usual sense. For every engine I've built, I wrote the rendering system from scratch each time. On one hand, that wasn't easy; on the other, I got to experiment with graphics to my heart's content. Whatever shape your rendering system ends up taking, sooner or later you're going to need to display text, for the UI, for debug info, and so on. You could, sure, draw letters by hand pixel by pixel, but it's better to grab a proper font library, like FreeType. For the adventurous, Unicode-loving types, there's HarfBuzz.

Resources

One day you finish up another level, launch the build... and it takes a minute to start. Then two. Then you notice the scene weighs in at 2 GB, because every single tree is an object with its own unique copy of the texture and model. And that's the moment it hits you: it's time to seriously tackle your resource system. Resource management is the layer players never see directly, but it determines how alive and responsive the game feels. The first sensible step is packing resources into archives. Opening a thousand files off disk is a huge drag on OS interaction. The platform might be slow, the file system might not be especially responsive, and the player's SSD might not be what you were counting on back when you were designing the game in the studio. So everything gets packed: textures, models, sounds, levels, scripts: all into archives. A custom container format with an index and compression is a common choice, but some people reach for ready-made solutions like Oodle, pack things into a ZIP, or build a virtual file system tailored to their engine's quirks. Big commercial engines, by the way, usually go exactly this route. Archives are easy to load, easy to cache, and nobody goes digging around in there by hand... except the people who really want to. You could, for instance, compile all resources into one big binary blob and "bake" it right into the executable: plenty of games have done exactly that. Xbox and Sony both did this early on. The upside? Everything's right there, everything's fast: no "loading," no file paths, you just grab a byte array and treat it like a resource. Sometimes this genuinely saved the day on different projects, and let them pass certification. And if the binary's 700 MB, well, that's clearly the cat's fault, or the junior "code-wrangers" who went overboard with templates. But of course, there are downsides too. You can't change a single asset without recompiling the entire game. Even if all you want to do is change the color of one pixel in a texture, you have to wait for a build. It also makes it pretty much impossible to change the game's behavior without access to its source code. How are players supposed to add their custom hats if all the content is soldered straight into the exe? You can also just store resources as plain files in an assets/ folder next to the executable. Then the engine simply loads the file it needs when the level starts, assuming we

haven't worried about performance ahead of time. This method is ideal for development: fast, transparent, and you can jump into the folder at any moment, swap out a texture, and see the change in the game. It's also a genuine gift for modders (modders: people who change a game's resources, through official or unofficial means, to get some new behavior or visual effect): grab it, change it, add to it: everything's wide open. No matter how you end up storing your assets, you still need some way to turn them from raw bytes into meaningful data. That's where format libraries come in: `stb_image`, say, to turn a PNG into an array of pixels. Or various JSON libraries, to pull entity information out of JSON-like config files, assuming you haven't given up on JSON as a format for these files yet. At some point, the team realizes that a purely declarative config format doesn't cover every case, and that's when a VM (Virtual Machine) for *JS/Lua/Squirrel/AngelScript* shows up in the engine, technically scripting languages, but here used just for parsing configs. In most cases, teams either invent their own format or reach for a high-level language, which also ends up relying on a virtual machine or interpreter in the end. You could also (I'd actually say should) add support for hot-reloading assets as the second step after configs, so you're not restarting the game a thousand times a day. Change an image, a sound, or a config, and poof, it's already picked up live in the game. Magic? No, just one more reason the resource manager grows into a monster that the rest of the engine suffers under. But resources aren't just about development, they're also about shipping. So the system needs to be able to generate builds for different platforms. Windows, Linux, consoles, Steam Deck: each one has its own constraints and requirements around data alignment, formats, and packing rules. Some platforms won't allow MP3 files, others don't support a particular texture format. And finally, the trendiest and, at the same time, most important piece: support for auto-updates and patches. A player shouldn't have to download 10 GB just because you changed one texture. My build system computes deltas: which files changed, how to compress them, and where to place them. The launcher checks versions and only downloads what's actually needed. And if something goes wrong, it can roll back. You'd be amazed how grateful players are simply because the game "updated itself and everything just works."

Audio Output

This, in case you didn't know, is absolute hell when it comes to platform-specific APIs. Ugly, unintuitive, finicky: Xbox in particular just loves throwing exceptions if you write even one byte too many into a buffer, and... well, if you write less than the buffer size you allocated, it'll throw a fit about that too. I'm not a real "audio welder," meaning not an audio programmer, but even a small taste of this particular adventure was enough to teach me: if

all you want is to play a sound, brace yourself for some suffering. Or... you can grab a library that hides all of this nightmare behind a reasonable interface. OpenAL or SDL, for instance, lately I've been using the latter, because it's pretty forgiving of mistakes, even when you're doing something completely wrong. These libraries usually expect an audio callback from the engine: a function that the system calls (typically from a separate thread) dozens or even hundreds of times a second. On every call, you need to hand back some audio data. Want to just sit there in silence? Fine. Want to play one single MP3 in the background? Also fine. But in all likelihood, you're going to need "a little bit more": music, effects, voice, all in separate audio streams, each with its own volume, with smooth fades, no clicks, no jitter, with reverb effects on top... you get the idea. To make all of that work right, you need a sample mixer. Not a human one, of course (though sometimes that wouldn't hurt either), but an audio mixer: a library that blends a whole bunch of sound streams into one, carefully controls their volume, filters out harsh peaks, adds effects, and keeps the audio from turning into mush when everything's playing at once. Especially when a fight suddenly kicks off in your game, with 15 different unit sounds, fanfare, and tense battle music all going at the same time. A good engine usually builds its own audio mixer. It can be big and complex, with mixing and reverb, effects and post-polling, or it can be a simple four-channel thing plus background music, but the core idea is the same: the mixer works with abstract audio streams, lets you combine them however you like, and spits out one final stream that gets sent to the audio output. It's probably worth writing your own at least once, just to understand how it works under the hood. Or you can go look for a library that already does all of it: mixing, balancing, compressing.

Physics

Honestly... most games don't actually need physics at all. I mean real physics: forces, impulses, friction, the whole academic toolbox. Most games need something simpler: animations, logic, scripts: things that create the impression of physics while having almost nothing to do with it. Take Civilization VI, for example: one of the most complex strategy games out there today. But it doesn't need physics. None at all. Or various city-building sims (SimCity, say), plain and simple, buildings don't collapse, roads don't crack, and nobody's computing the fall vector of a water tower. Simply because none of that's needed. Why make it harder than it has to be? And yet both Civ and Cities: Skylines, for some reason, drag in a PhysX dependency, and maybe they don't just drag it in, maybe they actually use it for something. In a platformer or a top-down 2D RPG, "physics" usually boils down to moving the character around and some simple collisions with walls, enemies, and crates. You can write that in a couple of evenings, assuming

you don't get stuck obsessing over every single line, the way I did while trying to figure it all out. The upside is that everything stays under your control: want the hero to slide around like they're on ice? Sure, go for it. Want enemies to bounce off each other on collision like a bunch of ridiculous rubber ducks? Also go for it. And if there are too many collisions and things start to chug, you can optimize: spatial hashing, quadtrees, and capping the range around the MC (Main Character, though some days "Main God" feels like the more accurate translation), the player the whole game basically orbits around. But... if you're stubborn and you genuinely know you need real physics, grab an existing library: Box2D for 2D, or Bullet for 3D. Don't reinvent the wheel: you'll have plenty of wheel-reinventing to do elsewhere in a homegrown engine as it is. And when you need to understand how engines like these actually work under the hood, your best bet is to watch the videos from Box2D's own author on YouTube. He goes into great detail about how constraint-based physics works through impulses, and gives a genuinely rare look under the hood. I used to think physics was about mass, velocity, and formulas. Now I know real physics is trying to figure out at 3 a.m. why your character is stuck on a staircase whose collision mesh never loaded (collision mesh: a special shape, or set of simple shapes, that describes the visible model of a player or object for the purposes of physics simulation).

AI

Out of every aspect of game development, this is my favorite topic. Over the years I've tinkered with monsters so many times that if I got a dollar for every one that got stuck in a corner, I'd own a house by now. First it was lone enemies, then swarms, then squads coordinating through shared BTs (Behavior Trees). And it turns out not every game actually needs "smart" AI. Yes, that sounds strange, but in half the cases, a couple of simple timer-driven scripts that yell "Oh!" and run screaming into a wall are enough to keep the player happy. Because AI isn't magic: it's just code. If your engine can write `if`, `switch`, run coroutines, or set timers, congratulations: you already support AI. All of "intelligence" is just a pile of conditionals cleverly disguised as a mind. But when you want to go pro, developers usually reach for one or several of the approaches covered below.

...Behavior Trees (BT)

At its core, this is just a hierarchy of tasks: "If hungry, eat; if you see an enemy, attack; if wounded, flee." BTs are a hierarchical decision-making structure, organized as a tree of composite and leaf nodes, where every node returns one of three statuses: Success, Failure, or Running.

Composite nodes (Selector, Sequence, Parallel) control the execution logic of their child nodes, implementing a top-down, priority-based task system. A Selector might hold conditions like "Health < 20% → Flee," "Enemy in sight → Attack," "Hunger > 80% → Eat," where each branch only runs when its matching condition is met. Leaf nodes (Action, Condition) implement specific actions or state checks. This architecture gives you naturally readable code: the tree literally mirrors the decision-making logic, which is exactly why BT-based approaches are so popular in game development, and why they lend themselves so well to visual editors with hot-reload support, letting game designers iterate on AI without recompiling. But! A BT with 500+ nodes turns into spaghetti that nobody understands, not even the author (especially not the author).

...GOAP (Goal-Oriented Action Planning)

"I want to kill the player. What actions would help me do that?" That's how a GOAP-driven AI thinks. It's not just following a script, it's building its own plan based on the actions available and the current state of the world. Pretty? Yes. Elegant? Very. But writing your own planner is genuinely hard. You have to account for resources, conflicts, action costs, cycles... GOAP is great for complex systems and big studios, but it takes a lot of time and a strong team to get it all working right. Unlike rigidly scripted behavior trees or finite state machines, GOAP takes a declarative approach: the developer defines a set of available Actions, each with its own Preconditions and Effects, and the planning system dynamically finds the optimal chain of actions from the current world state to the desired one. For a goal like "kill the player," for instance, the planner might build a plan like "pick up a weapon → approach the player → attack," automatically figuring out along the way that attacking requires a weapon, getting a weapon requires reaching the armory, and moving requires actually being able to move. This produces more natural, adaptive behavior, where an NPC can react to changes in its surroundings by replanning on the fly.

...FSM (Finite State Machine)

This is the classic approach. States and transitions: idle → walk → attack → flee. An FSM (finite state machine) is a mathematical model where an object can be in exactly one of a finite set of states, and transitions between states happen according to specific rules when certain events or conditions occur. This pattern is a perfect fit for describing the behavior of characters and enemies. A player might sit in states like "idle," "walking," "running," "jumping," "attacking," or "dead," while an enemy AI might cycle through "patrolling," "chasing," "attacking," "retreating," or "seeking cover." Each state runs its own specific logic: animations, sounds, physics parameters, available

actions. Transitions between states are governed by clear-cut conditions (a key press, a collision, a change in health, distance to the target) which makes the behavior predictable and easy to debug.

...Utility AI Picture an NPC with a whole set of available actions, each one eligible to run once enough of its trigger conditions line up. The system constantly recalculates the weight (utility) of each available action and picks whichever one scores best, producing smooth transitions between behaviors without the hard conditional jumps of an FSM or the heavy planning overhead of GOAP. But once an NPC has 10+ different actions, each with a utility function depending on dozens of parameters, it becomes nearly impossible to predict how it's going to behave in any given situation. Debugging turns into a nightmare ("why did the NPC suddenly start dancing instead of attacking?"), requiring you to trace through every utility function and how they interact. The need to fine-tune weights for different enemy types and situations tends to clash with locally optimal decisions, which is why Utility AI systems often get paired with an FSM for high-level states, or use hybrid approaches where the utility functions only operate inside specific contexts, keeping their unpredictability in check.

...the exotic stuff: HTN, ML, neural nets, RL HTNs (Hierarchical Task Networks) extend the GOAP concept by using hierarchical task decomposition, where complex goals get broken down into subtasks through a system of methods and primitive actions. This is a neural network capable of playing your game as a specific NPC, building a structured plan with room for long-term strategic thinking. Reinforcement Learning (RL) and neural networks promise even more impressive results: agents that learn optimal strategies on their own through interaction with their environment, adapt to the player's individual style, and display self-organizing (unique, emergent) behavior. Research projects have shown genuinely impressive results: AlphaStar in StarCraft II, OpenAI Five in Dota 2: agents that outperform human players in complex strategy games. These achievements create the illusion that this technology is ready for mainstream use, but training an agent for even a relatively simple game can take millions of runs, which in real-world time translates to weeks or months of continuous computation on powerful GPU clusters. Training a single agent can cost hundreds of thousands of dollars, and any change to the gameplay or any new mechanic means retraining it from scratch. There's also the problem of how unpredictable a trained model's behavior can be: not even the ML engineers who built it can really debug that "black box." As a result, even large studios tend to limit ML approaches to specific niche tasks (procedural generation, player analytics), while relying on proven classical methods for core gameplay AI, where behavior is predictable and development costs are under control.

...pathfinding

The A algorithm and its many variants are the workhorse of game AI, giving you cheap pathfinding through a graph of nodes via a heuristic function. Classic A uses the actual cost of moving through the graph's nodes from the start to the current node. Various tweaks to the algorithm turn what's theoretically a fairly involved mechanism into a practical tool capable of handling hundreds of navigation requests per frame. The main challenge is building and maintaining the discrete navigation graph and mapping its waypoints back into the game's actual space. A hands back a sequence of points, but the NPC has to physically move between them, taking into account its own size, momentum, obstacles, and dynamic objects. The classic problems include: getting stuck in corners because of oversimplified geometry, logical deadlocks when trying to squeeze through narrow passages, the inability to avoid moving obstacles (other NPCs, the player, dynamic objects), and issues with vertical navigation (stairs, platforms, jumps). But despite all of that, A remains the most reliable navigation system out there, even though every new map still needs careful geometry tuning and edge-case debugging.

...navmesh

And finally, the foundation of navigation itself: the navigation mesh (navmesh) turns a world's complex 3D geometry into a discrete network of nodes and links, where every polygon represents a walkable area, and the edges between polygons represent possible transitions. Building a good-quality navmesh means balancing representational accuracy against computational cost: too detailed a mesh creates excess nodes and slows pathfinding down, too coarse a one leads to unnatural paths and corner-sticking. Modern tools automate the generation process: NavPower/Recast/Detour (the open-source standard used in Unity and Unreal Engine) analyzes a level's geometry and builds a navmesh based on the agent's parameters (radius, height, max slope angle, jump height); some solutions offer more advanced algorithms for tricky scenarios, while big studios' proprietary solutions are tuned for their own engine's specific needs. But even the most sophisticated generation algorithm doesn't understand the level's semantics: it has no idea (and it's genuinely hard, algorithmically, to tell a building corner from the edge of a lattice-walled wooden gazebo) that a certain area is meant for stealth, that a bridge can collapse, or that a staircase needs a special climbing animation. Typical problems include: creating links through thin walls or windows, incorrectly connecting platforms at different heights, ignoring dynamic obstacles (doors, elevators, destructible objects), and broken graphs in geometrically complex areas. That's why modern

development combines automatic generation with manual "finishing touches" from a level designer, who places special markers for jump points, cover spots, and corner-handling, and tags areas with different properties (movement speed, traversal cost, restrictions by unit type). Without that manual navigation polish, even the smartest AI systems turn into helpless "cats with no paws": perfectly capable of making smart decisions, but physically unable to carry them out in the game world.

...AI isn't about "smart enemies"

It's about the illusion of intelligent behavior that lives entirely inside the player's head. What matters is that the enemy reacts, applies pressure, and occasionally surprises you. Under the hood, it's always just if, switch, and a chunk of map data, but the player doesn't know that. And let them not know. Because good AI isn't always "smart" AI. AI is, first and foremost, about keeping the player entertained and not letting them get bored, not about intelligence. Let the NPC be dumb: what matters is that it behaves the way the player wants it to. Let it be predictable: that actually makes patterns easier to learn, and tactics only benefit from it. Let it not be "real AI" at all, just a well-written script: as long as it runs reliably, entertains the player, and doesn't "glitch out." AI isn't about intelligence. It's about the illusion of intelligence. And if you manage to convince the player that something smart is playing against them, you did your job right. The only catch is that a smarter player might get bored exactly where a less experienced one is having a blast, simply because the AI is too simple for the former.

Scripts and Configs

Scripting languages like Lua, Python, JS, or Angel are very often used for embeddable logic, especially when the engine itself is written in C++. They're a lot more forgiving of runtime mistakes and can save you a ton of time. Even if you're building an engine from scratch, it's easier to plan for scripting support from day one, taking into account the capabilities of the language the engine itself is written in, which cuts overhead and simplifies the integration. Building an engine still requires programming: there's no shortcut around that. Which language should you pick? Honestly, it's not critical: the best language is the one you already know. An experienced developer can optimize performance-critical spots far better in a language they know well, and avoid the typical beginner pitfalls along the way. Low-level languages like C++ give you transparency in managing resources, but they also demand a much deeper understanding of architecture and development theory.

... it's not about performance, and it's not about FPS Scripting systems and config files exist first and foremost to speed up designer iteration and lower the barrier to entry for non-technical team members. High-level scripting languages like Lua, Python, or a custom in-house one give you a flexible sandbox for quickly prototyping gameplay logic, letting designers experiment with balance, build new mechanics, and tune AI behavior without having to rebuild the entire project or pull in a programmer. With low-level languages, every single change requires a full compile-link-deploy cycle, which can take anywhere from a few minutes to a few hours depending on the size of the codebase. Using interpreted scripts and JSON/XML configs creates a clean architectural split between a stable C++ engine core and ever-changing content. This approach does add a bit of runtime overhead (parsing, interpretation), but it dramatically cuts down development time and makes it possible to roll out A/B testing, mods, and patches without recompiling the game's binaries.

... it's about convenience and development speed Scripting enables a fundamentally different development model: the "write it, test it, fix it" loop shrinks down to seconds, thanks to the language's interpreted nature and built-in hot-reload systems. You can tweak an enemy's search algorithm or its aggression parameters on the fly, and see the result right there in the live session. Modern engines offer live variable inspectors, debug consoles, and visual debugging tools, which becomes essential given how constantly designer requirements change. Being able to build and test several AI behavior variants in a single evening gives the team real flexibility to experiment. Unlike a C++ pipeline, where every change costs anywhere from 30 seconds to several minutes of recompiling, a scripting environment lets developers stay in flow.

... surprisingly, it's also about security Sandbox architecture in scripting systems gives you an isolation mechanism, where each script runs inside a controlled environment with a restricted API. Lua VM, for instance, lets you create separate states (lua_State) with their own custom tables of global functions, letting you block access to the file system, network operations, or system calls. This is done through a whitelisting approach, where scripts only get access to explicitly approved functions, like `SpawnActor()`, `GetPlayerHealth()`, or `PlaySound()`. You can set a max execution time for a script, cap its memory use, limit how many function calls it can make, or restrict recursion depth, and abort execution once any of those thresholds gets crossed. If something does go wrong, the interpreter can catch the exception, log the error, and let the main engine keep running without a critical failure. Native C++ modules or DLL plugins, on the other hand, run in the same address space as the main application, where a bad pointer or a buffer overrun will take the whole thing down.

... it's about making the whole project reconfigurable Values baked straight into C++ source code, with no easy way to change them, pile up architectural debt: every tweak to an AI parameter means a full project rebuild: preprocessing, compiling, linking, deploying. This is especially painful when tuning things like damage multipliers, pathfinding weights, aggression thresholds, or reload times. The right approach is to move every tunable value out into external config files (JSON, XML, INI) or database tables, so the game's behavior can change without touching the source code at all. Modern engines offer reflection-powered property systems, where parameters load automatically from configs and can be changed through runtime inspector tools. Script-reloading tech works through a mechanism that watches files and dynamically reloads code, where the interpreter keeps an eye on file changes and reloads modified modules without stopping the main process, letting you update AI logic, event handlers, or gameplay mechanics in real time.

... it's a door into the community User-generated content needs a low barrier to entry and minimal technical requirements for modders. Exposing a C++ API means you have to ship an SDK with header files, build documentation, and support for a pile of different configurations (Visual Studio, GCC, various versions of Clang). Scripting languages solve this problem through their interpreted execution model, where the entire needed runtime is already baked into the game, and all it takes to create content is a text editor and a basic grasp of the language's syntax. Modularity built around a scripting API lets you define clean, well-specified interaction interfaces, where the API itself stays stable even as the internal implementation changes. Unlike C++ mods, which need recompiling with every engine update and can introduce internal bugs, scripted content just keeps working, as long as the API itself stays the same.

... there's no pleasing everyone here

Lua has become the "gold standard" for embedding in game engines thanks to how simple and efficient it is. The language takes up almost no space, runs very fast, and can be wired into C++ in literally a few minutes. You'll find a huge amount of documentation and example usage online for it. It works great both in single-threaded apps and in multithreaded processing, which makes it a solid, universal choice for most projects. *Python* has rich capabilities and a comfortable syntax, but integrating it into a game engine can turn into a real headache. Spinning up the Python interpreter takes real effort and a deep understanding of the language's internals. This seriously limits what you can do with multithreading, which is why a lot of developers end up switching to Lua instead. *JavaScript*, in the form of embeddable engines, delivers excellent results in both execution speed and memory

footprint. Modern JS engines are well optimized and can make for a solid alternative to the more traditional choices. Languages like *AngelScript*, *Wren*, *Squirrel*, and *ChaiScript* occupy a niche slot in the market. Capability-wise, they sit somewhere between the simplicity of C and the flexibility of Lua. Had these languages shown up before Lua became widespread, they might have made for a worthy replacement, but these days, it's hard for them to compete with an already-entrenched standard. C#, integrated into a game engine (say, by embedding the CLR, or through middle layers like Mono/Unity), gives you a powerful "scripting" layer with familiar OOP and static typing. It's a great fit for larger teams: there's existing infrastructure for reflection, debugging, unit tests, and hot code swapping. The downsides are a noticeable startup cost and the need to wrap calls across a managed/unmanaged bridge, which complicates integration and increases memory use. **daslang**: an interesting language that Gaijin built specifically for convenient embedding into the company's own projects. It was designed from the start for tight integration with C++: it supports static typing, compiles to bytecode, and makes it easy to expose engine functions and structs. Thanks to its strict type system and minimalist runtime, it gets you close to "native" speed, and it has a surprisingly well-developed library ecosystem inside the War Thunder project. C++**Script** (sometimes C++/CLI Script) offers scripting syntax that's as close to C++ as it gets. That lets you keep using familiar language constructs and reach existing C++ types directly, with no extra wrapper layer. It's a good choice when full compatibility with the engine and access to low-level capabilities matter most, but its computational and memory model stays just as complex as C++ itself. This fits experienced teams who want a scripting layer without giving up control over performance.

Networking

It's no secret that everyone dreams of this: synchronized worlds, co-op, PvP, MMOs, or at the very least a chat between players. And then you open Wireshark and start dreaming of just one thing: that all of it simply works. Let's start with the good news: maybe your game doesn't need this at all. If you're making an offline game, a puzzle, a platformer, some kind of "SimTower with frogs," you can forget about networking entirely. But if you've gotten it into your head to "add multiplayer," brace yourself. You're stepping onto a path of pain where everything depends on the details. There's no such thing as a universal networking architecture.

Modern languages (C#, Rust, Go) already come with pretty decent socket abstractions: async calls, nice coroutines, all that good stuff. In C++ (my condolences), you'll probably have to reach for something third-party like RakNet/ENet/UNet: libraries that aren't exactly cutting-edge, but are reliable

and have proven themselves over time across a lot of projects. Let's be honest with ourselves here: sockets are only the beginning. Connecting to a server is maybe 1% of the work. That's where the real fun starts. What even is a "networking engine," anyway? Good question. I asked myself the same thing while writing my first co-op prototype. I never did find the answer, but I ended up with a whole lot of questions about the functionality covered below.

...state synchronization This is the foundation of any multiplayer game, and at the same time, one of the hardest parts of network programming. Every player sees the game world through the lens of their own client, but all of these "worlds" need to stay in agreement with each other. The problem is that perfect synchronization is physically impossible because of network latency: by the time information about one player's action reaches everyone else, enough time has passed that the game situation might have changed completely. In practice, developers use different synchronization approaches depending on the type of game. In turn-based strategy games, you can afford to wait for confirmation from every player before moving to the next turn. In real time, you have to make trade-offs: either slow the whole game down to match the "slowest" player, or predict actions and correct for mistakes afterward. A third approach is an authoritative server that makes all the calls by itself, though then players with high ping end up significantly limited in what they can do. Things get especially tricky with fast-changing states: say, when two players shoot each other at the same instant, or several players try to grab the same item. The system has to figure out who was actually first, accounting for each player's different connection latency. Get this wrong, and you end up with a "rubber-band" game, where players' actions keep getting undone and recalculated.

...input prediction and rollback netcode Rollback netcode is a technology that lets games run smoothly even under high network latency. It works on the principle that the game predicts other players' actions and keeps simulating forward without waiting for confirmation. When the real data finally arrives, the system compares it against its prediction, and if there's a mismatch, it "rolls back" the game state and recomputes everything from scratch with the correct data. Implementing rollback requires storing multiple snapshots of the game state and being able to restore any of them quickly. On every frame, the game saves the full state of the world, including every object's position, velocity, animation state, and other parameters. Whenever a rollback is needed, the system loads the state from the right point in time and recomputes every event that's happened since. With a correct rollback implementation, the player barely notices any of these rollbacks and recalculations happening under the hood. The game stays

responsive even at 100-150 ms ping, which is impossible with traditional approaches. But building a system like this means rewriting nearly all of your game logic to be deterministic and rollback-capable.

...interpolation, extrapolation, smoothing

Networked games are constantly dealing with the problem of uneven data arrival: packets show up at irregular intervals, sometimes get dropped, sometimes arrive out of order. Without extra handling, this would result in a jittery, unplayable mess. Interpolation solves that by creating smooth transitions between known object positions. If the system knows a player was at point "A" at time T1, and at point "B" at time T2, it can smoothly move them between those two points. Extrapolation takes this a step further and tries to predict where an object will be at the next moment in time, based on its current speed and direction. The catch is, the further out the system extrapolates, the higher the chance of being wrong, since the player might suddenly change direction, or just stop.

...a secure communication protocol Network protocol security starts with basic validation of all incoming data. Every message from the client needs to be checked for correctness: is it in the right format, do the values stay within acceptable bounds, does the player actually have permission to perform the requested action. Simply trusting client data wide open a door to a whole pile of exploits, anywhere from "teleporting" a player to any point on the map to instantly granting them every resource in the game. Authorization and authentication protect against unauthorized users connecting and against identity spoofing. Every session needs to start with a procedure that verifies the player's identity, typically through tokens or digital signatures. It's important to refresh these tokens regularly and check their validity on every critical action. Session keys provide an extra layer of protection, even if the main credentials get compromised.

...encryption (nobody becomes paranoid for no reason) Encryption in games isn't paranoia: it's a grim necessity. Even casual games transmit data that shouldn't fall into the wrong hands: player scores, in-game currency, personal information. Modern traffic-analysis tools are so easy to use that pretty much any kid can intercept and decode unencrypted packets. TLS/SSL encryption has become the standard way to protect data in transit, but it adds noticeable latency to every packet. For real-time games, that can be a dealbreaker, since an extra 10-20 ms for connection setup and per-packet processing is enough to ruin the experience. That's why a lot of developers use lightweight encryption algorithms, or only encrypt the data that's actually critical, leaving the lower-priority stuff unencrypted.

...object replication Every object in a multiplayer game exists in multiple copies at once: on the server, and on each player's client, several versions simultaneously. The server-side unit holds the authoritative data on health, position, and inventory, while the client-side copy might show a tentative version of that data, adjusted with prediction and interpolation. Replication systems have to decide which object properties to sync, how often, and for whom. Private information like inventory contents should only go to its owner, while public information goes to every player within view range. Delta compression and other optimization tricks help cut down on the amount of data sent, but they also make the synchronization logic more complicated.

...logging and debugging Logging in networked games isn't just a technical nicety: every player action, every change to an object's state, every network packet needs to leave a trace in the logs. The problem is that the sheer volume of this data grows fast, since a popular game can generate gigabytes of logs in a single hour. The system has to be able to filter out what matters and compress or discard the less critical data. Structured logging with timestamps and unique session IDs lets you reconstruct the exact chain of events that led to a bug. When a player complains that their unit suddenly turned into the wrong type, the developer needs to be able to trace that object's entire history: when it was created, what commands it received, how its properties changed over time. Without detailed logging, bugs like this turn into unsolvable mysteries.

...the fight against cheaters Cheaters mean a permanent war with no final victory in sight, and developers don't always come out ahead. Every new defense spawns new workarounds, every anti-cheat patch provokes the development of even more sophisticated tools. The problem is made worse by the fact that the game's code runs on the player's own machine, over which they have total control. Any data that reaches the client can, in principle, be modified or reverse-engineered. Server-side validation remains the main line of defense, but it can't be made airtight without turning the game into a slideshow. Anti-cheat systems scan the player's memory and running processes looking for known cheat tools, but that raises legitimate privacy concerns. Players understandably don't love installing admin-level software that can monitor their entire system. Anti-cheat tools also frequently clash with completely legitimate software, from OBS screen recording to antivirus programs.

UI

You can make a game without physics, without AI, even without audio. But without a UI, sorry, that's pretty much impossible. We live in a world where a game with no UI gets written off as cheap junk. And somewhere, you do

need to write "New Game" and "Exit," right? On paper it all looks simple: no big deal, just draw a couple of buttons, some text, maybe a slider or a checkbox. Except these objects "suddenly" turn out to have state. Every UI element has to correctly handle a whole variety of user interactions. Buttons, text fields, and other components need to react, both visually and functionally, to idle, hover, pressed, focused, and disabled states. Managing the UI's hierarchical structure adds another layer of complexity on top. Modern UIs are built around nesting: a button can sit inside a panel, which in turn sits inside a window or a dialog. That window might stay hidden until the user activates the matching tab or performs some specific action. This kind of multi-level structure demands careful management of visibility, accessibility, and event handling at every level of the hierarchy. On top of that, add the different coordinate systems you have to juggle when building a UI. Unlike 3D graphics, where objects live in one shared world space, user interfaces operate in screen-space coordinates and coordinates relative to their parent element. That means computing element positions, handling clicks, and every other operation all have to account for the quirks of 2D screen space and its transforms. Universal input support is another important piece of the puzzle. Users expect an interface to work equally well with keyboard and mouse on desktop, touchscreen on mobile, and game controllers on consoles. Every input type comes with its own quirks: mouse precision, keyboard navigation convenience, the natural feel of touchscreen gestures, and controller ergonomics. Finally, to make a UI genuinely appealing, you have to add visual polish in the form of animations and smooth transitions between states, animated element entrances and exits, and visual feedback on interaction, none of this is just decoration, it's what makes an interface feel alive and responsive. And just like that, you're no longer making a game: you're writing a UI framework, one that, frankly, probably won't get used anywhere outside this one specific game. And even if you grab something decent (Scaleform, you're not as bad as you first seemed), you'll still have to wire it into your engine, hook it up to the input system, the event system, shaders, text rendering, and on and on. UI is a complex, structured, multi-layered, platform-dependent, and deeply underrated part of any engine. But you'd better go through this at least once, even if just at a bare minimum, done properly, without trying to outdo Unity UI, Unreal Slate, or HTML.

Tools

When it comes to auxiliary tools for working with an engine, as you've probably already guessed, not every engine or game actually needs them. Simple Python scripts, for instance, can work fine as tools. Most engines ship with full-blown visual editors out of the box, something like a hybrid of an

IDE and Blender, letting you place objects in a scene, edit materials and shaders, configure physics and interactions, preview animations and UI, and test everything live. But building a proper editor is a task roughly as complex as building the engine itself. For a small engine, building a Unity-style editor isn't necessary, and honestly isn't worth it: it would be inefficient and eat up an enormous amount of time. That said, don't dismiss the simple tools that take just a couple of evenings to write but make life dramatically easier.

...level converters Custom binary data formats give you optimal load performance and memory footprint by using structures that match the engine's internal representation directly. Unlike text formats like JSON or XML, which require parsing and type conversion at runtime, binary data can be loaded straight into memory with minimal overhead. This matters a lot for levels containing thousands of objects with positions, transforms, materials, and metadata. A custom format also lets you bake in, wherever it makes sense, data compression, precomputed data structures, or navmeshes. Conversion utilities can hook into the file system, rebuilding game assets automatically whenever source files change. Separating source assets from packed ones lets you change your storage format transparently without disrupting the content team's workflow, and lets you roll out optimizations for different target devices.

...asset validation Asset validation systems are an important part of the build pipeline, catching potential errors and keeping the content consistent. A validator checks dependencies, since every asset can reference other resources through an ID (GUID), a path, or its own reference system. It also verifies that materials have their textures, models, animations, and shaders in place. Hooking validation into CI (Continuous Integration, a continuous build-and-test system) lets you catch problems at the asset-build stage, before they ever make it into the final build, and generate reports pointing at the exact issues. This becomes especially important on large projects, where an artist might have no idea their change broke dependencies somewhere else in the game, and QA ends up with crashes with no clue what caused them. It's worth remembering: time spent building a tool should save you more time down the road. Don't build a visual editor just for the sake of having one. But if you find yourself hand-editing 200 lines of JSON every single time, a simple homegrown program or script will save you a ton of time.

Architecture

At some point, you've got everything built separately: graphics, audio, assets, physics, UI, maybe even AI and scripting. But one important thing is still missing: how do you wire all of this together into something that actually works? There are a few approaches that actually hold up.

...no architecture is also an architecture

A lot of indie developers, tired of the complexity of modern libraries, fall back on a simple library-based approach. The idea is to build a handful of independent subsystems, each responsible for one specific piece of functionality. You create a gfx subsystem for rendering, audio for playing sounds, assets for loading data off disk, and input for handling user input. Each subsystem runs in isolation, with no idea the other parts of the program even exist. You, the developer, decide yourself which functions to call and in what order. The whole process stays entirely in your hands. This approach, by the way, was quite popular in the late nineties and early 2000s. This method has some real upsides: it's simple to implement, transparent in how it works, and a great fit for small projects. But its success depends heavily on the team's personal discipline, since there are no automatic scene managers or other supporting structures to lean on. Over time, it gets harder and harder to hold the whole call graph and its cause-and-effect relationships in your head. At first, everything works smoothly and looks great, and for a lot of people, that's enough. Later on, of course, you'll run into bugs like the UI never getting notified that an asset updated, or the background music failing to "kick in" after loading a new level. Fixing problems like that is the inevitable price you pay for having no connections between different parts of the program. And those connections, in essence, are exactly what application architecture is: the very thing we were originally trying to avoid.

...the god object: also an architecture Big game engines like Unity, Unreal, Godot, CryEngine, Dagor, and others often go with an approach built around a single base class. It's usually a class named `Object`, `Entity`, `GameObject`, `Actor`, or something similar, which becomes the ancestor for every game entity. That base class usually holds virtual methods for the core game functions:

```
struct GameObject {  
    virtual void update(float dt);  
    virtual void render();  
    virtual void onEvent(const Event&);  
};
```

Every game element you create (UI buttons, enemies, NPCs) inherits from this single class. Each of them automatically gains the ability to update on every tick of the game loop, render to the screen, and respond to various events. Convenient? Very. You get a uniform interface for every game object right out of the box. A centralized structure like this makes it easy to manage game entities, and lets you build a global object manager. The main problem is relying too heavily on inheritance: every class descends from a single

parent, which runs against the "composition over inheritance" principle. Things get difficult the moment you need an object to support multiple behaviors at once: the system becomes brittle and hard to extend. A couple of months in, `GameObject` turns into a giant 300-line dumping ground of virtual methods. Trying to fix it with components only makes things worse. Eventually you start to suspect this isn't a great design, but for a lot of people, it's good enough to ship a game.

... ECS: it's trendy! But...

This architectural model has become quite popular in recent years among indie developers and enthusiasts. The core idea is a radical separation of data and logic within the game system: everything is built around three key concepts. An Entity is just an identifier, carrying no logic or data of its own. A Component is a container for one specific kind of data, like an object's position, its health pool, or sound parameters. A System is a functional block that processes components according to certain rules. None of these work on their own, but together they let you throw out the traditional object hierarchy entirely, replacing it with batch data processing. A physics system, for example, can grab every Transform and Velocity component at once and update the positions of every moving object in a single pass. The upsides here are high performance thanks to efficient use of the processor cache, a clear separation of responsibilities between the different parts of the system, and solid scalability. On the downside, you're committing fully to a complex system that often demands a huge amount of boilerplate code. Without specialized editors, ECS code is pretty much unreadable, and without visual debuggers, you can't really debug it either. For small-scale projects, this is like using a tank to swat flies.

Game-Engine

So maybe you should just leave it as-is? Yes and no. The engine, and everything around it, isn't the goal: it's a tool. If you're making a weekend hobby project, don't try to build the perfect framework, because all you'll end up with is a framework, and the moment you start writing a framework, actually making the game slides to the back burner. But it's also worth remembering that architecture is like the game's skeleton. Nobody sees it, but if it's crooked, everything else is going to ache, bend, and break at the slightest sneeze. And when things get really uncomfortable, don't be afraid to throw it all out and start over. Don't be afraid to throw away something broken, everyone does it (the rule of three systems). On top of that, GEA is still just as relevant as ever, and nobody's come up with anything better yet, if you ever get "stuck," go reread the matching chapter in this book. To start understanding how games actually work, you don't need an engine on the level of Unreal. To avoid becoming yet another "sort of" Unreal C++ programmer, who knows containers exist but has no idea how they work inside, go build your first (or second) game without an engine. Make something primitive that opens a window, draws a couple of sprites, and has a tiny resource manager that just packs textures into a zip. Let it be a game-engine: simple, buggy, full of "reinvented wheels," but yours, and more importantly, built with an actual understanding of the tools involved. No Unity, no Godot, none of the other convenient solutions where you skip past 90% of what's actually happening under the hood. In exchange, you get to tinker with architecture, design patterns, and genuinely understand resource optimization. This path takes time, but in the end you come away with more than just a game: you come away with an understanding of the "inner workings," and it's hard to overstate how much that knowledge is worth for your career down the road. This is where real professional growth starts, the genuine kind, with no shortcuts. Someone who comes into the industry only knowing Unreal can only really work in Unreal, while a programmer who understands how it all works under the hood can work in Unreal, in Unity, in Godot, or in any other engine. But... if Unreal Engine, Godot, or some other ready-made solution feels more comfortable to you, why not? Even if you end up settling on someone else's engine, the experience of building your own will make for a solid foundation for your professional growth, teaching you to see the magic of games behind the technical reality. *The choice, as always, is yours.*

OX11

Game Engine Architectures

Before diving into game engine architectures themselves, I think it'll help to say a bit about what I mean by software architecture and how it connects to games.

First off, architectures (plural) exist, whatever people in the games industry like to claim. Second, it turns out there's more than one of them. That'll help explain why the chapters are laid out in this particular order. Either way, the next time you get dragged into an argument about how terrible (or, conversely, how brilliant) some specific game engine and its architecture is, you'll have a couple of arguments and a sense of what's actually going on. Very often, what happens is: we write the code, the editor, the game, the project's skeleton gets fleshed out with "meat," and that's when the problems nobody paid attention to catch up with us, because we had to ship something that at least looked like a working version. But just because we ignored those problems and "swept them under the backlog rug" doesn't mean they stopped being problems.

Every program has an architecture. Even if everything is crammed into a single `main()` and "sort of works," that's an architecture too. But it's a lot more interesting to figure out what actually makes a game engine good.

Nobody writes an engine "just because." It's born out of necessity: while making a game, you end up having to formalize and wall off its individual parts so they can be reused later. That's how an editor, a resource-build system, a separate I/O module, a dedicated renderer, and a bunch of other things gradually come into being. Eventually a team of engineers forms around all of this to maintain those systems. Usually this only becomes clear after the game ships: that's when it sinks in that working the old way just isn't an option anymore.

Second option: the game itself is the editor, the engine, and the build system, one that can package itself up as finished levels and logic. Every game engine and editor out there grew out of a game: some games (and the engines/editors that followed) became successful, and on the wave of that success the developers decided to get themselves extra time and resources to keep developing it. By that point, a team usually ends up with someone (who grows into the role, or more rarely gets hired into it) whose job is to define, refine, manage, and keep an eye on the components inside the game-engine, making sure they work together and the whole system stays coherent. You can call this person the "Architect," though more often they're the studio engineering team's resident "Elder," the one who ends up doing the most legwork with QA, programmers, designers, and everyone else.

Every working day for the last decade or so, I've been staring at the code of various games and various engines, mostly big ones that have been in development long enough to grow up a bit.

Of course, like any programmer who's spent a long time in one domain, I've developed a sense for good design, or at least, I'd hope, a nose for bad design. Every developer has had that moment where the code in front of them was so bad that the best thing to do with it was bury it respectfully in a comment and write the whole thing again right next to it. "Best," though, doesn't mean "approved by management." Old, terrible legacy code is usually at least reasonably well tested, and everyone already knows where the landmines are buried, whereas with new code, you still have to go find them yourself. Few of us get lucky enough to work with genuinely well-designed code: the kind of code, project, or engine that feels like a luxurious '70s American sedan: it's got everything, you just need to know where to find it, and it's ready to hit a hundred fps the moment you put the pedal down. And working across different codebases over the years, I started noticing that game engines end up looking a lot like the teams that write them.

Once a team grows past five or so developers, people start organizing and coordinating their work, and that often means splitting up tasks and responsibilities based on each team member's technical strengths. Everyone does what they know best: the internal database, the renderer, physics, the editor, tests, and so on. This split looks perfectly logical from an organizational standpoint: everyone's in their lane, which helps spread the load and boost efficiency. But there's a catch: splitting up the knowledge this way means different parts of the system can't communicate effectively with each other, and development gets harder, because specialists end up working in isolated subsystems, not talking to each other about things that actually matter. This artificial division ultimately makes it harder to share knowledge and leads to weak integration between the game engine's components.

Melvin Conway spotted this pattern between an engine's structure and the team's internal organization back in the 1960s. Conway's Law states that the organizational structures behind a system's design inevitably end up shaping that system's architecture. In simpler terms, organizations struggle to build systems that are structurally or functionally independent of how their people communicate. And this isn't just about game engines; it applies to anything a team builds together. When programmers and architects design a system, they unconsciously reproduce the same divisions in the code's structure that already exist inside the organization. If a dev team is split into separate groups (one working on the front end, another on the back end, a third on the database), the system ends up riddled with those same divisions, becoming a mirror of how we communicate inside the studio, company, or team. If a system's architecture calls for tight integration

between different parts (gameplay and rendering, say), then instead of forcing those teams to work in isolation, you need to actively organize direct interaction between them.

This can mean forming joint groups, pair programming, or cross-reviews, which leads to better knowledge sharing and, in turn, changes the architecture itself.

Note

To fix a project's architecture, Jonny LeRoy proposed changing the team's internal communication instead. The idea is that building a more harmonious architecture requires evolving the structure of teams and the organization alongside the system's architecture. Instead of building a system that mirrors organizational "weaknesses," you need to change the organization itself so that it encourages a more integrated, flexible system. !!! From here on, the architecture labels used in the text aren't applied with any strict criteria and can be debatable in places, since they're based on personal hands-on experience, porch conversations with colleagues, and feedback from other projects and studios. !!!

ox12 Unity (Unitary Architecture)

When I first ran into Unity (really into its architecture and internals, not just the editor's looks), it was, to put it mildly, a shock. My expectations for a game engine that had conquered the mobile world were things like sane architecture, some kind of development plan. Instead, I got a patchwork quilt of components with three separate build systems (as of 2014; things may have changed since, though I doubt it), all duct-taped together with a mono VM just so it would run, held up by a pile of platform-specific hacks and comments like "Don't remove this blank space," "Don't compile this on Independence Day," or "Build with this constant first; if it doesn't compile, set it to 0." Building the editor required its own wiki page with 117 carefully numbered steps: what to do and in what order, which libs to build first, which ones to rebuild again at step X, which flags and options go where, and who to pray to if it all fails anyway. And building the editor had its own separate build system on top of that. But if you look at how Unity actually came to be, some of these questions start to make a lot more sense. It all started with a game of their own called GooBall, which students David Helgason, Joachim Ante, and Nicholas Francis set out to build in 2004. Their studio, Over the Edge I/S, ran into the typical problems indie developers faced in the early 2000s: expensive engine licenses, complicated code, and no resources to speak of. They finished the game (it shipped in 2005), but it didn't sell particularly well. What came out of that game's source code, its approach, and that same team, though, was Unity. In 2005, Unity 1.0 was quietly shown off at an Apple conference. You'd think an engine built for

Mac OS X wouldn't stand much of a chance in a world where almost everyone games on Windows. But that ended up being its edge: Mac OS was popular with designers, and a little later it became the go-to choice for the first wave of iPhone developers. Unity stood out immediately. Instead of complicated code, it offered a component-based approach: you could assemble objects like LEGO bricks, bolting on physics, animation, or scripts. A visual editor (a rarity in the mid-2000s) let you simply drag and drop elements with the mouse. And support for C# plus a simplified UnityScript made scripting accessible. The industry took notice fast. The turning point came in 2008, when Unity launched on Windows, growing its audience tenfold. But the real breakthrough came two years later, with the launch of the Unity Asset Store, a marketplace for ready-made models, textures, and scripts, where most of the junk cost a buck or less, if it wasn't free outright. That was manna from heaven for indie studios and solo hobbyists alike. Around the same time, the engine added iOS and Android support, which lined up perfectly with the smartphone boom. Suddenly, anyone with a laptop and an idea could build the game of their dreams. That's how we got Temple Run (2011) and Monument Valley (2014). A year later, in 2011, EA, Blizzard, and Ubisoft all took notice, signing long-term support contracts and licensing it (effectively buying the source outright, with exclusive rights) for internal use, and by 2015 nearly half of all mobile developers were using Unity. Under the hood, it was all still that same patchwork quilt of libraries, three different build systems, a whole junkyard of reinvented wheels, with a partially modified EASTL, Boost, and the standard C++ library all mixed together, and different parts of the engine could be using different STL implementations, which meant you often had to flat-out copy data around, say, between a renderer that "lived" on EASTL and an editor built on custom classes and containers. On top of that, bolt on a Mono runtime nailed in sideways, which you also had to shuffle data with, and you get a classic early-2000s game-engine architecture. To avoid sounding too harsh, I'll call this kind of architecture "Unitary." There's already a more precise term for projects like this in English. But remember the saying that there are only two kinds of programming languages: the ones everybody complains about, and the ones nobody uses. A similar comparison probably applies here too, but I genuinely like this engine a lot, so don't let this section put you off it.

ox13 Big Ball of Mud

This style of writing code with no clear structure is called a "Big Ball of Mud," an anti-pattern popularized by Brian Foote in the mid-1990s, though I've also heard colleagues use related terms like "cup of mud," "fancy barrel," and "dirty feet." In modern terms, a "Unitary architecture" might describe, say, a simple game where event handlers call straight into the processing

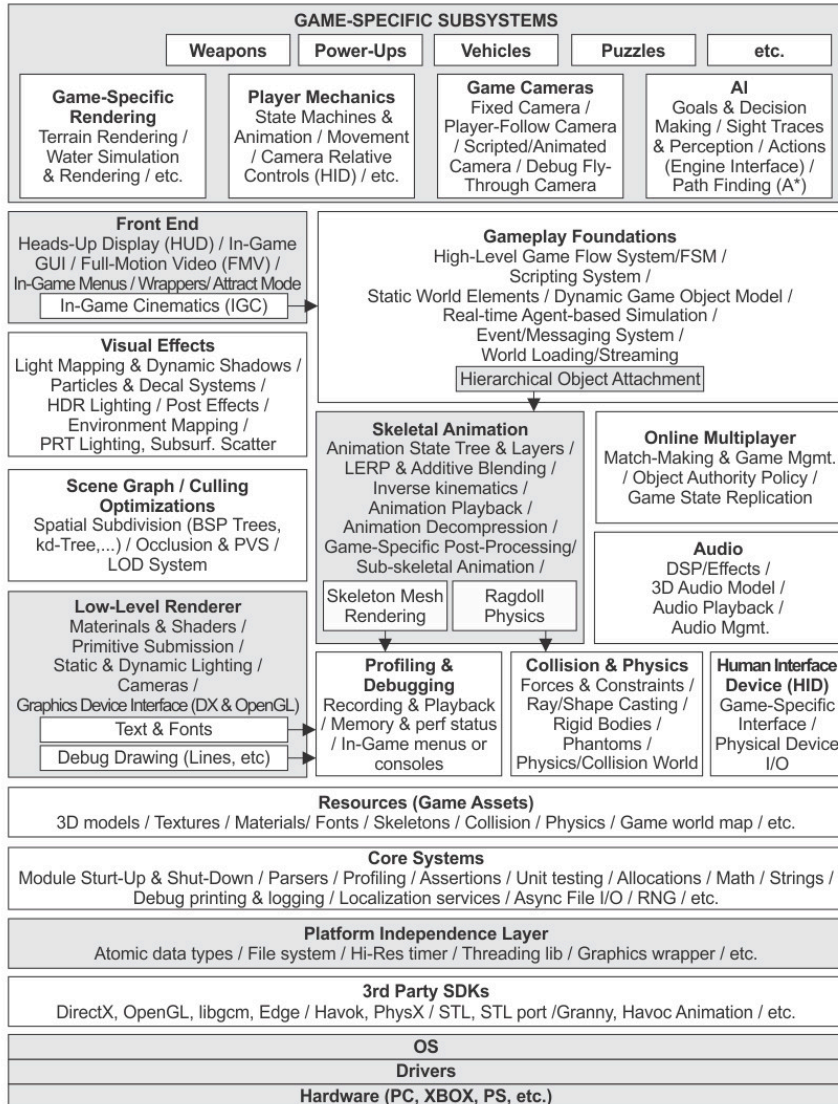
logic, with no internal buffering or dispatching of any kind. Most games start out this way, and it eventually turns into an unmanageable mess as the project grows. For a game with three mechanics and a couple of screens, it doesn't matter, but the lack of structure eventually makes every change harder and harder, and the system itself starts to suffer from problems shipping new versions, testability, scalability, and performance. This anti-pattern shows up everywhere. Almost nobody plans to build a "Unitary architecture" on purpose, but plenty of projects slide into it from a lack of control over code quality and structure. In a system like this, any change to one class causes unpredictable side effects in others, turning every bit of extra work into a nightmare. In the worst case, by the end of the project you get a "mess" of source files, resources, and the engine's own technical files all tangled together. What started as a three-student project is now maintained by a team of 400 engineers. Unity Technologies as a company has around 5,000 employees, fewer than 10% of whom are programmers. Figure 2.1 shows a diagram of Unity Engine's components, where everything looks decent, even pretty good, from the outside... *Right up until you dig into the engine's C++ guts and try to change something.*

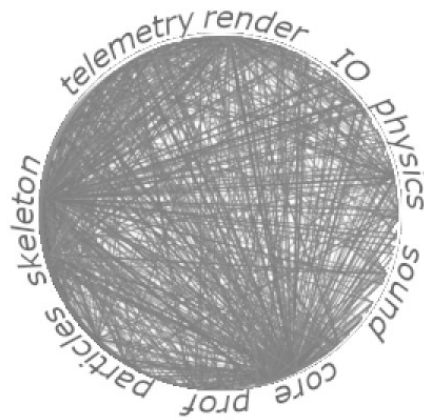
Note

A "Big Ball of Mud" is a haphazard, structureless mass of incompatible components glued together in a hurry. Systems like this show clear signs of unchecked growth and hasty fixes. Data gets passed indiscriminately between different parts of the system, often to the point where nearly all the important information ends up global or duplicated. The system's overall structure may never be clearly defined in the first place. And if it ever was, it can get eroded beyond recognition over time. Developers with even a shred of architectural instinct steer clear of swamps like this. The only people willing to work on systems like these are the ones who don't care about architecture and are fine putting up with the daily grind of patches that barely hold the chaos together. (Brian Foote)

By this point, it's clear the "simple" architecture stopped being simple a long time ago. Everything is tangled up with everything else: managers depend on other managers, states reference each other, events fire off unpredictably. Every attempt to improve something starts breaking other systems, and just to change the logic of a single action, you end up having to rebuild the dependencies of half the project. And the longer all of this goes without a refactor, the scarier it gets to even approach the code. What's worse, in a structure like this, the cost of building new features skyrockets, since they trigger conflicts where you can't even tell "what affected what" or where things are going to break. Components that were never meant to be reused turn into traps for new programmers. There are no clean boundaries, no clear dependencies, no isolation, so there's no reliability either. That makes developers afraid to touch anything, and the project settles onto a

plateau of stagnation. The first figure shows the overall diagram of the Unity engine as described in its official documentation (version 4, 2014), and the second shows an actual class-connectivity diagram for the engine, where every chord represents a data relationship between the different classes and subsystems shown in the first figure.



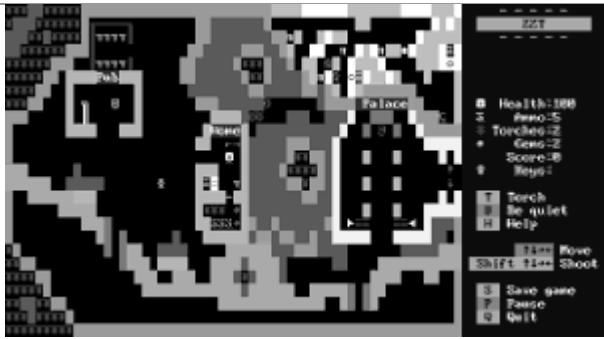


The *"Cup of Mud"* (*legacy swamp*) architecture consists of localized pockets of chaos inside otherwise reasonably well-organized code: individual functions, classes, or modules that are small islands of mess inside a fairly orderly system. Code like this usually comes from temporary fixes that were never cleaned up, or from attempts to quickly patch a critical bug. A function might end up with dozens of parameters, a pile of nested conditionals, and mixed logic from completely different levels of abstraction. While the scale of the problem is smaller than a full *"Big Ball of Mud,"* these localized pockets of mess still become a steady source of bugs and slow development down, since developers avoid touching those parts of the code. The term *"Fancy Barrel"* describes code with an attractive outward-facing interface or API that hides a chaotic, poorly organized implementation underneath. On the surface it shows off good documentation, elegant method names, and modern design patterns, but the internal architecture stays tangled and unstructured. It's a nice-looking facade hiding a barrel of technical debt and creating a false impression of quality: the problem shows up the moment you need to extend the functionality or fix a bug, and all you're left to enjoy at that point is the pretty interface. The terms *"Dirty Feet"* and *"Pigsty"* describe code that's tracked dirt all through the system, leaving messes and contamination scattered across the project. It's marked by broken encapsulation, where changing one module forces you to modify a bunch of other, seemingly unrelated components. The people who have to work with it end up having to touch several different parts of the system just to implement one feature, which blurs the logic and creates implicit dependencies. The end result is code that's hard to test and even harder to isolate, since every component drags along a pile of side effects and needs a complicated setup just to work with.

ox14 Unreal Engine (Layered Architecture)

In 1991, Tim Sweeney, founder of Epic Games (back then still just Epic MegaGames), started building editing tools for his first games. It all began with an adventure puzzle game called ZZT, with very simple graphics. The goal was to fight various creatures and solve puzzles in maze-like, top-down levels. Using plain ASCII characters to draw characters, enemies, and environments, ZZT ran in DOS text mode. At the time, the game wasn't considered revolutionary in terms of graphics or gameplay, but Tim's approach to programming, and especially the built-in ZZT-OOP level editor, laid down the modularity principles that would later evolve into Unreal Engine. ZZT turned out to be a success, bringing in around \$100 a day for its author through shareware sales. That was enough to convince him that making games for a living was actually possible. After hiring a small team out of the ZZT level-design competition's participants, he moved the business out of his parents' house, and the five-person team got to work on the next project: *Jill of the Jungle*. In 1992, Epic released *Jill of the Jungle* (fig. 2.4), a DOS platformer that introduced more advanced tools like sprite animation, movement physics, and particles. But the real highlight was the emphasis on a component-based approach: Tim used data structures built on standard engine components, letting you override object behavior without rewriting the whole codebase. You could create new enemy types, tweak their AI, or add interactive level elements through scripts. While most studios were writing code more or less "from scratch" for every project, Tim's approach split the game into an "engine" (low-level rendering, physics, and audio systems), "content" (assets and basic logic), and "scripts" (config files and high-level logic).





This approach later carried over into Unreal Engine, announced in 1998 alongside the game Unreal (fig. 2.5). That very first engine-and-game combo (Unreal) was already built from independent components: shared logic, a renderer, the UnrealEd level editor sitting on top of it, and a separate scripting language, UnrealScript, object-oriented, with class inheritance, which made modding a lot easier.

Other features included collision physics with dynamic objects, 16-bit color, dynamic lighting from up to three sources, and the ability to launch the game straight from the level editor. The game was a massive success, selling more than 1.5 million copies. In 1999, Epic released its second game, Unreal Tournament, which was essentially a "bugfix release" that added built-in network support to the engine. The second version of Unreal Engine debuted in 2002 with America's Army, a free multiplayer shooter built to boost recruitment for the service. It was the first time the military had deployed large-scale game technology for something other than internal use, and the game even picked up a few awards from players. Even though PC games remained Epic's main focus, Sweeney and his team once again saw where the market was heading: toward consoles. Building a single engine that would let developers ship games on PCs as well as the Sony PlayStation 2, Nintendo GameCube, and Microsoft Xbox looked extremely appealing. Unreal Engine 2 also pushed Sweeney's interest in modularity further: instead of one stable release, it became a constantly evolving product that kept getting updated and modified after its first release in 2002.

Flexibility became another key element of Unreal Engine 2. While earlier engines from various companies were mostly "locked" to a single genre (first-person shooters), Epic's team realized that giving developers the freedom to work across different styles would become a serious competitive advantage.



Integrating the Karma physics engine, a particle system, and a bunch of other new features quickly turned UE2 into an industry standard. It powered successful games like *BioShock* and *Thief: Deadly Shadows*. Unreal Engine 3 shipped in 2006, with the first *Gears of War* as its flagship title; new platforms joined the supported list, and it became possible to ship games on iOS and Android. This update was aimed more at the end user: the internal architecture stayed largely unchanged, but players got a dramatically improved renderer with per-pixel lighting, better physics, and a lot more besides. Layered architecture is one of the most common ways of organizing code in game development. Even though Unreal Engine was shaped for a long time by one particular person's style, it still holds up for large teams, keeping the engine and its parts structured and scalable. If you look at how the connections inside the engine are actually wired up, yes, different parts still talk to each other, but there are far fewer of those connections than there could be. Layered architecture here fits naturally on top of a component system: every component, whether it's a scene object, physics, or scripts, forms its own layer of abstraction. All of that effectively becomes the standard approach and the foundation for the architecture of most studios' own games built on this engine. Under Conway's Law, a codebase's structure usually mirrors the team's own structure. This is especially visible with Unreal's development: the team splits into small groups: UI and input, gameplay programmers, AI engineers (so you can build NPCs, objects, and monsters), and a large team of engine optimizers. That same organization carries straight over into the teams building the actual games. Individual developers can also pick up tasks from different areas or rotate through other groups; that's part of the company's broader policy. But people are still people, and developing on Unreal is pretty relaxing, so teams sometimes

slide into a "default architecture," just following the engine's templates and established practices without really thinking about why. It gets worse when a "Unitary architecture" starts creeping into a project and developers just "start coding" with no clear plan. For a while, they'll unknowingly end up following the layered approach the engine carries with it, but without proper structure and support, everything eventually ends up as a "Big Ball of Mud." As of 2023, Epic Games employs roughly 4,000 people worldwide, across teams working on games, publishing, the store, and the engine itself. Various open-source estimates put the number of people working directly on the engine somewhere between 500 and 1,000+. Besides the roughly 200 programmers doing graphics, physics, and optimization work, the company also counts in this number technical-support engineers who maintain the GitHub repo and review incoming commits, technical writers and course engineers, platform developers (VR/AR, mobile), and a large division developing Virtual Production, CAD integration, and work with major game studios and film studios (another roughly 400 people).

ox15 CryEngine (Microkernel Architecture)

In the early 2000s, as developers kept looking for ways to build ever more realistic and immersive game worlds, a unique game engine showed up on the scene with a fundamentally new take on architecture. CryEngine, built by the German studio Crytek, turned out to be revolutionary not just for its stunning graphics, but for its innovative microkernel architecture too.

It all started with a small tech demo called "X-Isle: Dinosaur Island." The tech demo, built by what was still a tiny team at the time, caused a sensation among publishers and developers. Back then, most game engines were built on monolithic architectures, with tightly coupled components, but the Yerli brothers (Avni, Faruk, and Cevat Yerli) had a different vision. Inspired by the design concepts behind the QNX operating system, they carried microkernel architecture principles over into a game project, and the result was the foundation of an engine that would later become one of the most technologically advanced of its era. Investors and studios bought into the approach once they saw that CryEngine isolated only the critical functions in a central core, with everything else running as separate modules. That modularity let you swap components out without risking the stability of the whole system: at one point, there were even implementations that let you hot-reload individual logic and linked libraries right in the middle of a running game, letting you update NPC behavior or fix logic bugs with no restart, where only truly critical errors would actually crash the game. And even then, there was a safe fallback handler that could reconnect the crashed component and let you keep playing. On top of that, this approach let teams develop new features in parallel and debug and optimize subsystems

without having to rebuild the whole engine. The brothers pushed the idea of building games across different genres on one shared technology base, but that particular ambition never quite panned out, and the engine ended up staying "the best engine for FPS games."

The first commercial game to show off the power of this microkernel approach was *Far Cry* (2004), which stunned players with its huge open spaces, draw distance, realistic vegetation, and a world that could adapt dynamically to the player's actions. The second version of CryEngine, used in *Crysis* (2007), effectively became the de facto tech benchmark for new generations of graphics cards, setting the bar for graphical performance, and the phrase "But can it run Crysis?" became a meme among players.

This particular engine's microkernel approach left its mark on the whole industry, bringing concepts like scalability, fault tolerance, and adaptability to new platforms and architectures into wider use. It also influenced other engines, which started borrowing bits of its approach for their own development. After the second *Crysis* shipped, scalability and adaptability became standard practice across game engine development in general. At the time it came out, CryEngine, like every other graphics engine relevant at the time, was cross-platform. The renderer supported *OpenGL* and *DirectX 8/9* and was built around open spaces: a single level could hold locations up to 3 square kilometers. The physics system was built in-house, and that didn't hurt its quality one bit: it supported rigid bodies, fluids, vehicles, cloth simulation, soft bodies, and ragdolls: everything implemented at the highest level for an in-house technology. Inverse kinematics let multiple animations be layered onto a model at once for more believable behavior. NPC AI ran on a Lua module instead of C++, with the nice property that scripts never needed recompiling. All in all, this engine gave us *Far Cry* with its many expansions, as well as the MMORPG *AION*. NCSoft's developers licensed CryEngine for their MMO, though they later reworked it heavily to fit their own needs, optimizing it for the specifics of massively multiplayer online games.

In 2002, after the engine's release and *Far Cry*'s launch, the shader module gained support for shader version 3.0, and the engine's version was bumped to 1.2. Shaders made a whole range of visual effects possible: per-channel lighting, bumpy reflections, refractions, smoke, animated textures, transparent computer screens, windows, and bullet holes. One standout feature was its fog implementation, which turned out to be far more realistic than anything other engines of the time could manage. In March 2006, Ubisoft acquired all rights to the *Far Cry* game, along with a license to the specific CryEngine build used in it. Ubisoft got usage rights to the entire *Far Cry* intellectual property: the trademark, the logo, the characters, the story,

the franchise, the setting, along with the right to keep using and modifying that version of the engine's source code, meaning the Crytek team could no longer use that particular codebase to build their next engine.

From that point on, Crytek had no legal ties to that codebase or to the Far Cry trademark, though the company kept the CryEngine name and the rest of the technology for itself. After Crytek shipped its last patches for *Far Cry*, the engine split and developed down two separate paths. With the rights to *Far Cry* now in hand, Ubisoft went on developing its own variant of the engine: for *Far Cry Instincts* and *Far Cry Instincts: Evolution* on Xbox, *Far Cry Instincts: Predator* on Xbox 360, and *Far Cry Vengeance* for the Nintendo Wii.

As Louis-Pierre Farand, lead producer of *Far Cry 2*, said in an official interview, the Dunia Engine uses only 2-3% of the original CryEngine's source code.

Crytek, for its part, kept developing its own technology base and built a new engine, CryEngine 2, which became the next evolution of the original CryEngine, completely independent of Ubisoft. Like its predecessor, it was written in C++, though its main difference was a lack of cross-platform support. The only supported platform was Windows, with DirectX. At launch, CryEngine 2 was one of the most technologically advanced and photorealistic graphics engines around compared to the competition, setting new standards for visuals, physics, and dynamic environments. But its steep system requirements and its Windows/DirectX lock-in limited its audience to owners of powerful gaming PCs. It kept, and even improved on, everything the previous version could do: *CryEngine 2* handled huge open spaces just as well, and on top of that it gained a resource-streaming system that let levels fill in as the player progressed.

Crytek's developers were among the first to move to a 64-bit architecture. Since level sizes had grown considerably since the original Far Cry, the resource cost of rendering them had grown right along with them, to the point where 32-bit systems just couldn't keep up anymore. The microkernel didn't go anywhere either: every additional module lives in its own DLL file and gets loaded as needed during gameplay. The compiled engine is spread across separate DLL files, with every engine component living in its own library file, which lets you swap components in selectively, modifying and replacing only the ones that need it instead of the entire engine at once.

A new physics engine, CryPhysics, was also built in-house at Crytek, designed from the ground up for multithreaded processing and fully integrated into CryEngine 2. That let the engine apply all kinds of physics

manipulation to practically every object in a level, trees and vegetation included, producing realistic reactions to gravity, wind, explosions, friction, and collisions with other objects.

The move to the third version, as before, came with its share of internal conflicts on the team and at the management level. A completely different group ended up handling development, so the engine can both pleasantly surprise you and let you down. CryEngine 3 was officially announced on March 11, 2009, and shipped that same year on October 14. The first game built on the new engine was the first-person shooter Crysis 2. The engine itself was originally written with DirectX 11 support, but the first game released on it ignored that API entirely, along with DirectX 10, until a patch came out that fixed the situation. The company most likely did this because it wanted Crysis 2 to be cross-platform, since the biggest downside of PC exclusivity was that every legitimately purchased copy came with roughly 10 pirated downloads attached, which meant consoles got special attention this time around.

Modern CryEngine 3 keeps building on the ideas behind the microkernel architecture, while increasingly leaning on AI for tooling, and of course for graphics. Even so, the philosophy the Yerli brothers laid down left its mark on the games industry, showing that modularity, flexibility, and scalability can go hand in hand with high performance. These days, the engine has effectively moved to an **open-source model** at <https://github.com/o3de/o3de>, which is an evolution of the Amazon Lumberyard engine, itself originally based on the 2015 version of CryEngine, whose source was separately published at https://github.com/CRYTEK/CRYENGINE_Source. Amazon's developers bought the CryEngine source, cleaned up the project's source files, and put them on GitHub, first as the licensed Lumberyard source.

Later, they handed it over to the community entirely for free, in the form of O3DE. By the time it went open source, more than \$50 million had reportedly gone into the engine, a serious sum, even if it's nowhere near what Unity or Unreal have absorbed over their much longer, much larger-scale development. The micro-modular architecture is now represented as independent "gem" components that you can freely add to or remove from a project. As of today, O3DE is still in active development, and while the engine is highly capable, it isn't nearly as widely used as Unreal or Unity.

0x16 Dagor (Data-driven Architecture)

Among the big-name game engines, mostly American-born, there are still a few technologies from Eastern Europe that have had a real impact on the industry. One of them is the Dagor Engine, built by Gaijin Entertainment. This engine stands out not just for its technical specs and its ability to run on pretty much any platform with a GPU and some kind of CPU attached, but

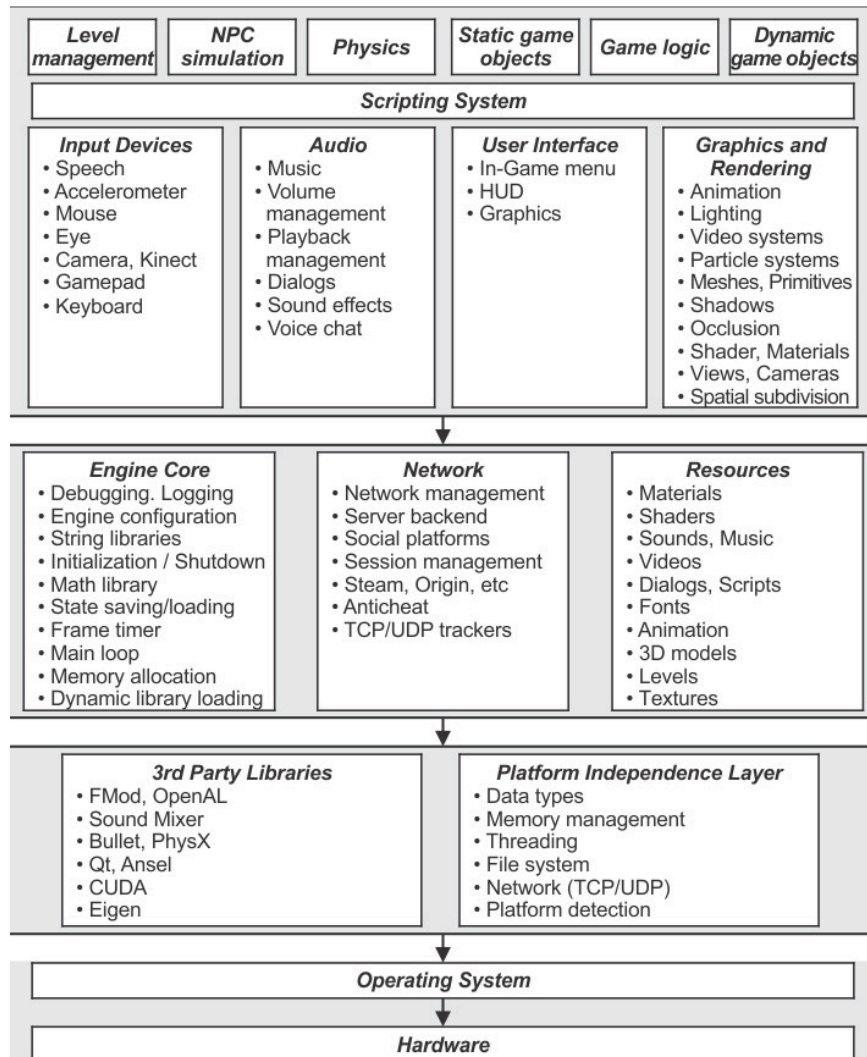
also for its unique architecture built around data-driven design. The engine is open source these days, so you can go look at the details yourself. This engine's history started in the early 2000s, when an internal game-dev tool became publicly known after the release of the flight sim "IL-2 Sturmovik: Birds of Prey" (2009), which showed off its ability to handle complex physics models and produce realistic visual effects, along with what data-driven approaches could do. Data-driven architecture is a software-design approach where an application's logic is defined mostly by data rather than hardcoded in the source. In the context of game engines, that means most of the game world, its objects, and their behavior get described in data files (often JSON, XML, or some other structured format) that the engine interprets at runtime. That comes with some real advantages: separating data from code, declarative definitions (objects, their properties, and their interactions are described declaratively: what to do, rather than how to do it), and dynamic configuration (changes take effect with no need to recompile code or even restart the game). All of these properties naturally grow into a component system that's defined and configured entirely through data. In traditional engines, changing a game object's behavior usually means writing or modifying code, compiling it, and then testing it; here, game designers and artists can tweak object parameters, effects, and even core behavior just by editing data files, what's called doing it "on the fly." During an internal tournament for one of the studio's games, the admins created random level objects live during the match just by dropping .blk files (basically a JSON equivalent) into the level folder, and the players ended up with cars, weapons, refrigerators, and a couple of tanks raining down "out of the sky." Data-driven architecture naturally lends itself to a component-based approach for designing game objects, and over time you start thinking in that paradigm by default and can't really imagine working any other way. Every object in the Dagon Engine can be assembled from a set of components, each one responsible for a specific slice of functionality. This makes it easy to create new object variants just by combining and tweaking existing components, and again, all of this happens "on the fly," with no need to recompile the engine and often no need to even restart the game. Despite the common belief that interpreting data at runtime slows a game down, a properly implemented data-driven architecture can still deliver excellent performance. You can see this indirectly just by watching the company's various games run at a solid frame rate on mobile devices or the Nintendo Switch, which is far from the fastest hardware out there. One of the cornerstones of a data-driven architecture is the resource system. All game data is organized as a hierarchical structure of configs, each with a unique identifier that can be loaded on demand. For the Dagon Engine, that might look something like this:

```
/data/  
  /vehicles/  
    /tanks/  
      /t-34/  
        model.blk  
        textures/  
          diffuse.dds  
          normal.dds  
          specular.dds  
        weapons.blk  
        collision.blk  
    /weapons/  
      /guns/  
        /85mm_zis/  
          ballistics.blk  
          Visual_effects.blk
```

.blk files are a special structured-data format used by this engine, functionally similar to JSON but optimized for fast loading, processing, and the ability to override sections and properties.

```
visual_effects.blk  
graphics {  
  rendInstDistMul:r=0.5  
  grassRadiusMul:r=0.1  
}
```

Despite all its advantages, a data-driven architecture also comes with some real downsides. The first is how hard it makes debugging, since behavior is defined by data rather than code, debugging gets a lot more labor-intensive, requiring new tools just to track how data changes affect system behavior, and it often means maintaining a parallel set of engine-specific visual debugging tools. The second is a performance hit from interpretation, even if the engine is optimized to handle data efficiently, interpreting that data still adds overhead compared to hardcoded logic. The diagram below shows the systems that can be configured just by changing scripts, with no need to rebuild the whole engine, and as you can see, that's pretty much the entire engine.



ox17 X-Ray Engine (Monolith Architecture)

There's no shortage of technologies in gaming history that were ahead of their time. One example is the X-Ray engine, built by programmers Alexander Maksimchuk and Oles Shishkovtsov for the S.T.A.L.K.E.R. series. The game was originally conceived as being about robots on some unknown planet with an anomaly zone and lasers, but the team eventually realized they didn't need to invent a planet or any robots at all: a zone full of anomalies and exactly the right atmosphere was sitting about three hours' drive from their office. First shown off back in 2001, this engine became the technological foundation for one of gaming's most atmospheric "universes."

The graphics were impressive for their time: high detail, up to 4 million polygons per frame, well above what even late-2000s games were pushing; large-scale spaces, since the engine handled indoor areas and open territories up to 2 square kilometers with equal efficiency; dynamic time-of-day cycles, a full day-night cycle with matching lighting changes; and weather effects like realistic rain, wind, and fog simulation. The dynamic lighting system in particular still produces memorable shots and an unforgettable atmosphere even today.

For physics simulation, X-Ray used the free Open Dynamics Engine (ODE), released in 2001. This open-source library provided rigid-body dynamics and collision detection, well-suited for simulating vehicles, creatures, and objects in a changeable world. In theory, thanks to its stable integration, the system shouldn't have randomly "exploded" for no reason. And yet, players of the first Stalker are intimately familiar with all sorts of physics anomalies (from flying bodies to objects behaving strangely) which turned into a signature quirk of the series, spawning memes and funny clips galore. But let's just call that a feature of the Zone; anomalies are anomalies, after all.

This engine is a textbook example of a monolithic structure (not to be confused with Unitary), split into parts, but with the connections between them kept to a tight minimum. Unlike other designs where systems run more or less independently, X-Ray is a tightly integrated system where every component is inseparably linked to the others. A monolithic architecture comes with both real advantages and serious downsides. Tight integration between systems (graphics, physics, AI) and the ability to build gameplay mechanics directly on top of component relationships let you use and plan resources more efficiently, through custom allocators, object pooling, fast message queues, and so on.

Direct interaction between components, with no extra abstraction layers in the way, delivered maximum performance on the hardware of the day. You can precisely control memory and CPU time, which was critical for such a demanding project. A unified architecture let the team realize a coherent vision for the engine, the game world, the dev tools, and the subsystems, where everything interacts naturally. Put simply, this is an approach to building software where the system's different components are tightly coupled and function as a single whole: you can't remove or split off any one part without losing functionality and slowing everything down.



But the very thing that makes it strong on performance also limits scalability, makes updating individual components harder, and causes real problems, especially on multi-core systems. And the complexity of evolving a system like this and fixing bugs in it grows exponentially, since changing one part drags along changes in its neighbors. Complexity like that ended up being X-Ray's biggest technical headache. Besides the random crashes every fan knows all too well (hello, "green bug"), another sore spot was the micro-stutters and hitching (frame judder) you'd see across every game in the series. They were especially noticeable in *Shadow of Chernobyl*, and in extreme cases the game could turn into a slideshow even while technically

holding a normal frame rate. The root of the problem was an engine architecture that leaned on the resources of just a single core, and that limitation became more and more critical as multi-core systems took over.

Officially, the last version of the engine is X-Ray Engine 1.6.02, used in *Call of Pripyat*. But the fan community didn't stop there: enthusiasts kept working on the tech, building unofficial versions of the engine that fixed plenty of critical bugs, added new features, and, most importantly, added support for multi-core CPUs and multithreading. Despite its technical problems, the engine remains a genuinely significant milestone in gaming history. Its strengths, dynamic lighting, deeply detailed physics, and the A-Life simulation system, were ahead of their time and gave fans more than a few wow moments, and maybe this particular monolith did end up fulfilling someone's dream of "the perfect game" after all.

Monolithic architecture, with all its advantages and drawbacks (and, more importantly, applied thoughtfully) is a good example of a particular approach to building game engines, one that's also a valuable lesson for the industry. The engine's concepts and the practices built around it later continued to evolve in a different series of games set in the *Metro 2033* universe.

ox18 Godot (Modules/Services Architecture)

From the outside, it might look like game development is moving at breakneck speed: new rendering approaches, neural networks for animation, skeleton and model-appearance changes, voice work. But under the hood, all of it still rests on tried-and-tested solutions that are extremely hard to break or change. I'd even go as far as saying that, outside of conference talks, game development is actually quite conservative: big studios don't want to take risks, and small ones simply don't have the money or the time to. Something genuinely new at the foundational level shows up pretty rarely. Against that backdrop, it gets especially interesting when concepts get dragged over from the web world, the way it happened with the Godot engine and microservice architecture. Most architectures get their names after the fact, either from the community or at conferences, where people notice a recurring pattern and start building it out further: there's no secret council of architects deciding what the next big movement is going to be. It's more that a lot of developers end up converging on similar solutions as an engine's ecosystem changes and evolves. The best ways of handling those changes and getting value out of them become the architectures other people imitate. Microservices are a bit different in that regard, since there was no engine or game that actually used these principles first: the term itself was coined and popularized by a blog article by Martin Fowler and James Lewis called "Microservices," published in 2014, where they laid out the shared traits of this relatively new

architecture. And it's hard not to see a parallel between its ideas and the way Godot ended up organized, even though Godot's node-and-scene model really traces back to an older lineage of GUI and scene-graph design, not to that specific article. The core idea is splitting a complex system into a bunch of small, autonomous services, each one responsible for one specific function. In the context of Godot development, this concept turns into a "microservice architecture," where the game isn't one monolithic "beast" of code, but a system of interacting components. Physics, AI, the user interface, the audio system: each of these can be built as a separate module with a clearly defined interface for interacting with the rest. I won't claim this approach looks ideal to me, but at the very least, it lets you focus on individual aspects of the game without worrying about how your changes will ripple into other parts of the project. The engine showed up on the scene in 2014 and has since won over plenty of developers with its simplicity and reliability. It was named after the famous character from Samuel Beckett's play "Waiting for Godot." What makes Godot unique is its approach to organizing game logic through a system of nodes and scenes. Picture a building-block set where every element of the game, from the character to the UI, is its own block you can easily snap together with others. While Godot wasn't originally designed with microservice architecture in mind, its node system and modularity ended up mapping neatly onto microservice principles. The end result is an architecture where different components can be developed in complete isolation from each other. Godot's signal mechanism gives you a genuinely interesting way to wire components together. Imagine the player picks up an item: the inventory system fires off a signal that other systems can react to: the UI updates its display, the achievement system tracks progress toward picking up items, and a unique-abilities system recalculates the character's stats, with none of these systems knowing any implementation details about the others, and maybe not even knowing the others exist. The core idea behind the engine, really, is that you add new functionality by building new modules rather than modifying existing ones. Component modules can be reused in other projects, and if a module has a bug, it's less likely to bring down the whole game, sure, it'll still have some effect, since a game can't be fully distributed the way a real distributed system can, but at least it'll crash less often. But keep in mind this architecture isn't a silver bullet either. For small projects, this approach works beautifully: you throw together some component modules, wire up a few signals, and voilà, everything just works. As a project grows, along with its connections and mechanics, everything starts piling up excess complexity and becomes counterproductive: the signal system turns into a bottleneck, and modules start fighting each other over updates. The thing to understand here is that using micro-modules only makes sense where they actually provide real value. Separating game logic

from visual presentation, for instance, is almost always worth it, while splitting tightly coupled mechanics into separate parts-as-services-as-modules will come back to bite you with needless complexity later. On top of that, unlike traditional web-dev microservices, which run as separate processes or even on different machines entirely, in a game all the components run within one process (or a small handful of them). That spares you from having to solve the interaction and synchronization problems that classic microservice systems run into, problems that would just get in the way more than help over such "short" distances. Even so, this approach has found its fans: more than a dozen relatively large games shipped on this engine in 2024 alone. That's nowhere near the thousands Unreal and Unity rack up, but it's a serious achievement for a project that's effectively run by five people and a community.

0x2

Low-Level Optimizations

0x20 Optimizations

Types of Optimization

Wikipedia defines "optimization" as the process of modifying a system to improve its efficiency or reduce its resource consumption. In the context of game development, that framing takes on a special weight, since every game exists under a tightly constrained budget. On one side of the scale: CPU time, memory, and frame time; on the other: what players and the market expect in terms of graphics, physics, AI, and overall user experience, and that scale almost always tips against developers. Optimization should always start with the big picture. There's no point spending time on random small tweaks: it's more important to find the real bottlenecks, where performance is actually being lost and which subsystems are burning more resources than they're worth. That's what profilers, performance counters, and tracing are for. The process looks like this: find the problem spot, measure its impact, try an improvement, measure again. If there's no improvement, roll back and look for the next candidate. The steps go like this:

1. **Test (Benchmark):** build a reproducible example of the problem situation.
2. **Measure (Measure):** establish the baseline memory and resource usage.
3. **Detect (Detect):** find the part of the program that's using resources worst.
4. **Solve (Solve):** change how it works to improve resource usage.
5. **Check (Check):** use measurements to determine whether there was an improvement.
6. **Repeat (Repeat):** go back to step one until you hit your performance goals.

Or a catchier version: *MEDAL*:

Measure->Examine->Develop->Assess(check results)->Loop An experienced developer treats optimization not as a bag of "tricks," but as a data-driven process. That said, it helps to have a set of proven techniques in your toolkit for memory, multithreading, rendering, math, and so on, applying your own experience and tools to get a measurable result without wasting time pointlessly. Unfortunately, there's no substitute for hands-on experience here: without good practice or mentors, it's hard to learn to spot problem spots. It's always amazed me how developers manage to pack huge amounts of computation onto relatively "weak" hardware, and what tricks and solutions they reach for to keep an application running fast. This isn't just true of game engines, either: it applies to databases, management

systems, and so on. The gap is especially visible when porting relatively recent games (PS3-generation and up) to various handheld consoles like the Nintendo Switch, Steam Deck, and mobile devices. The Nintendo Switch isn't all that far ahead of the PlayStation 3 in raw performance, and trying to port demanding games designed for at least the next generation (PS4) of consoles runs into serious problems that aren't easy to solve. Here's a comparison of three generations of gaming platforms' capabilities (table 3.1), useful for understanding what level of solution can move between platforms without major compromises, and for understanding which level of optimization you'll need to pay attention to when working.

Component	PlayStation 3	Nintendo Switch (handheld)	Steam Deck
CPU	Cell x1 PPE + x6 SPE ~25 GFLOPS	4×ARM Cortex-A57 1.02 GHz ~25 GFLOPS	AMD Zen2 2.4-3.5GHz (~448 GFlops)
GPU	NVIDIA (~GeForce 7800) ~192 GFLOPS	NVIDIA Maxwell GM20B ~157 GFLOPS	AMD RDNA 2 ~1.6 TFlops
RAM	256MB XDR 256MB GDDR3 ~25.6 GB/s (GPU)	4 GB LPDDR4 ~25.6 GB/s	16GB LPDDR5 ~88 GB/s
API	proprietary	OpenGL ES 3.2/Vulkan/NX	OpenGL3+/Vulkan

From a CPU standpoint, the PS3 uses the Cell architecture with one main PPE core and six active SPE coprocessors. In real game workloads, that configuration delivers something like 20-30 GFLOPS, though tapping into the Cell's full potential is notoriously hard. The Nintendo Switch has a 4-core ARM Cortex-A57 at 1.02 GHz, though it's really only a 3-core chip in practice, since one core is always reserved for the system, delivering roughly comparable 20-25 GFLOPS under typical load. Despite the architectural differences, both processors land in roughly the same ballpark. The GPU side looks a bit different. The PS3 has the RSX, an adapted version of the GeForce 7800 with peak performance around 192 GFLOPS and an outdated shader model. The Switch uses a GPU based on NVIDIA's Maxwell (Tegra X1): in handheld mode it runs at a reduced clock, delivering around 157 GFLOPS,

while in docked mode (plugged into the dock) performance can reach 393 GFLOPS. As for RAM, the Switch has 4 GB of LPDDR4 with around 25.6 GB/s of bandwidth, while the PS3 has 256 MB of XDR for the CPU and 256 MB of GDDR3 for the GPU at roughly the same speed. The Switch wins slightly on capacity, but not on bandwidth. The Steam Deck is included as an example of today's minimum hardware baseline that developers design around, meaning porting a level down is still possible, with big caveats, but jumping a level up just isn't possible for purely technical reasons. This three-way comparison is approximate; once the Steam Deck 2 ships, the Nintendo Switch effectively becomes the new PS3-equivalent, and the overall bar for specs rises. That doesn't change the fact that the levels of optimization stay the same, with the same problems and the same toolbox of solutions, just scaled up to bigger resource budgets. Either way, a game is pretty demanding software, often with soft real-time requirements (needing to hold a steady 30-60-120 fps), whether you're shipping it for the PS3 or the latest Steam Deck. You need to deliver acceptable speed no matter what, or the game will be uncomfortable to play and nobody will buy it. One small advantage when porting to mobile platforms is their relatively stable hardware, though the sheer "zoo" of processors, GPUs, and environments across smartphones actually argues the opposite. Consoles have it a bit easier here, with hardware specs only changing every few years. When it comes to porting a game, optimizations can be split into a few levels: architecture, algorithms, and code.

Note

When developing or porting a game, it's important to remember that it needs to run at a comfortable performance level on the target platform. To do that, you should optimize at three key levels: architecture, algorithms, and only as an absolute last resort, low-level code.

Architecture-Level Optimizations (~50% of the gain)

Architectural decisions cover the module interaction system, the data-layer stack, the choice of memory allocation scheme (custom allocators vs. system malloc, object pooling vs. free objects, dynamic allocation vs. allocation-free), and the threading model (single-threaded, multi-threaded, shared state, actor-based concurrency).

A suboptimal architecture shows up as excess memory consumption, slow execution, idle threads, resource contention, or expensive resource access. Fixing architectural flaws takes a lot of time and manpower to refactor the game's or engine's core subsystems, which often means rewriting core project components like object management, rendering, or the memory system. For example, moving from inheritance-and-components to an Entity-Component-System (ECS) architecture, replacing the object-allocation

mechanism, or reorganizing the command system. Changes like these deliver the biggest performance gains, since optimizing memory access can improve performance several times over, and properly organizing thread work can put every CPU core to use and cut down idle time. But architectural changes late in development create a domino effect across the entire codebase, requiring months of fixes and expensive regression testing, which makes this kind of optimization essentially impossible right before a game ships or when porting to a new platform.

Algorithm/Data-Structure-Level Optimizations (~30%)

Algorithms determine how efficiently your program runs. Picking the right algorithm can literally save your frame rate: replacing a plain linear scan of an array ($O(n)$ complexity) with a binary search over a sorted array ($O(\log n)$) or a hash-table lookup ($O(1)$) delivers a massive speedup as the data grows. In an array of a million elements, a linear search might take a million operations, a binary search just 20, and a hash table will find the element almost instantly. Sure, we rarely have arrays that big, but it only takes a handful of frequently called spots for that "necessary" million lookups to add up, in the bad sense. But an algorithm's theoretical complexity doesn't always line up with its real-world performance, because of how modern hardware works: processors use caches (L1, L2, L3), and algorithms that access data sequentially get a huge boost from hardware prefetching. As a result, a simple linear scan over a tightly packed array can easily outrun a hash-table lookup if the data fits in L1 cache. Linked lists, theoretically efficient for insertions, end up slower than dynamic arrays in practice because of "pointer chasing." Processors use branch prediction, but that only works well when the conditions in your code are predictable; a simple algorithm with clear logic can run faster than a complex one with a bunch of checks. Which is why developers often look past Big-O notation at the real-world conditions: data size, memory access patterns, branch predictability, the target hardware's architecture. Development involves profiling code against real data and building benchmarks for the tricky parts. It often turns out that a "naive" implementation beats the "theory" under specific real-world conditions, thanks to better memory locality and no overhead from maintaining a complex structure. Here's a classic example to illustrate the point:

```
// Example: optimizing a lookup
bool slow_contains(const std::vector<int>& vec, int
target) {
    // Calls find every time
    return std::find(vec.begin(), vec.end(), target) !=
vec.end();
}

bool fast_contains(const std::vector<int>& sorted_vec, int
target) {
    // A simple optimization
    // use binary search on a sorted array
    return std::binary_search
(sorted_vec.begin(), sorted_vec.end(),
target);
}
```

Source-Code-Level Optimizations (~10%)

Local optimizations (within a single block of code, function, or algorithm's scope) are about fine-tuning a solution to speed up operations and cut overhead. We're talking about small-scale tweaks that usually touch a relatively small part of the program and can be handled in a single code review: up to a hundred changes across up to ten files at a time. The main approaches include heapless programming, implementing algorithms and structures on the stack, object pools, and specialized allocators. Loop unrolling improves instruction-level parallelism when the compiler can't apply that optimization automatically because the code is too complex. Other techniques include inlining "hot" functions, bringing in SIMD for vectorizing computation, and paying attention to data locality and matching usage patterns. The optimization process often involves detailed profiling with tools like Intel VTune, PIX, Razor, Tracy, or a game engine's built-in profiler. Analysis involves looking at inefficient instructions, measuring cache-miss counts, redundant virtual function calls, hidden copies when passing parameters, and "expensive" operations on objects. Fixing these issues usually means trading off code readability against the solution you're introducing, so this kind of optimization only gets applied to important parts of the code that genuinely eat up a meaningful chunk of frame time. One of the most common tasks at this level is loop optimization. Nested loops that the compiler can't improve on its own show up constantly in code, and they can often be simplified or swapped for more efficient algorithms. For instance, if an inner loop is searching through a collection, it's worth

considering preprocessing the data structure or using a hash table to avoid redundant passes. Sometimes it's enough to hoist invariant computations out of the loop or change the traversal order to ease the load on the CPU:

```
void calculatePhysicsSlow(vector<float>& pos,
vector<float>& vel, int frameStep){
    for (int i = 0; i < iterations; ++i) {
        for (size_t j = 0; j < positions.size(); ++j) {
            // Invariant computations repeat
            // the compiler might not catch them in
            complex cases
            float gravity = world->getGravity();
            float damping = object->getDamping();
            float timeStep = getDeltaTime() / frameStep;
            vel[j] += gravity * timeStep;
            vel[j] *= damping;
            pos[j] += velocities[j] * timeStep;
        }
    }
}

void calculatePhysicsFast(vector<float>& pos,
vector<float>& vel, int frameStep){
    // Invariant computations hoisted out of the loops
    float gravity = world->getGravity();
    float damping = object->getDamping();
    float timeStep = getDeltaTime() / frameStep;
    const float gravityTimeStep = gravity * timeStep;
    for (int i = 0; i < iterations; ++i) {
        for (size_t j = 0; j < pos.size(); ++j) {
            vel[j] += gravityTimeStep;
            vel[j] *= damping;
            pos[j] += velocities[j] * timeStep;
        }
    }
}
```

Modern compilers with optimizations turned on can apply transformations like this automatically, but in complex cases involving pointers, virtual functions, or tangled logic, manual control remains the only way to squeeze out maximum performance. Hoisting invariant computations is a fundamental optimization based on the loop-invariant code motion principle, where expressions whose result doesn't change across iterations get moved outside the loop to be computed just once. Invariant operations

include working with constants, accessing variables that don't change, math expressions with fixed values, and calls to functions with deterministic results. For example, computing `deltaTime/iterations`, `sin(angle)` for a fixed angle, or `sqrt(maxDistance)` inside a loop leads to redundant work on every pass, whereas hoisting these into precomputed variables cuts the operation count from $O(n)$ down to $O(1)$. The next important step is finding and removing linear searches in your code, especially inside loops. A linear search (say, `std::find` or manually walking an array) can be perfectly fine for small collections, but inside "hot" code paths or with large amounts of data, it turns into a serious bottleneck. Swapping a linear search for a binary search, using associative containers, or presorting the data can speed things up significantly:

```
struct GameObject {
    int id;
    string name;
    float x, y, z;
    bool isActive;

    GameObject(int _id, const string& _name,
               float _x, float _y, float _z,
               bool _active)
        : id(_id), name(_name), x(_x), y(_y), z(_z),
          isActive(_active) {}
};

// Linear search by name - O(n)
GameObject* findObjectByName(const string& name) {
    auto it = find_if(objects.begin(), objects.end(),
                     [&name](const GameObject& obj) { return obj.name
== name; }
    );
    return (it != objects.end()) ? &(*it) : nullptr;
}
```

The main optimization techniques boil down to three approaches: hash tables, binary search, and caching. Hash tables (`unordered_map`) give you average $O(1)$ lookup complexity, and they're especially effective for frequent lookups by unique key. It's good practice to build indexes up front, say, mapping an ID to its position in an array or building an index by string name, which speeds up data access considerably. Caching lets you store the results of expensive operations and return them quickly on repeat requests. The key is keeping the data fresh and invalidating the cache correctly. A one-time investment in indexing and sorting pays for itself quickly once you're

doing repeated lookups, so it's important to pick the right data structure for the specific usage pattern and be ready to make trade-offs: faster access often costs extra memory. Hash tables are great for frequent key lookups, binary search for stable, rarely changing data, and caching for heavy but repeatable computations. All of this matters a lot in game loops, where objects get queried hundreds of times per frame. Here's one example:

```
void addObject(const GameObject& obj) {
    size_t index = objects.size();
    objects.push_back(obj);
    idToIndex[obj.id] = index;    // Update the indexes
    nameToIndex[obj.name] = index;
}

// Lookup by name through a hash table - O(1)
GameObject* findObjectByName(const std::string& name) {
    auto it = nameToIndex.find(name);
    if (it != nameToIndex.end()) {
        return &objects[it->second];
    }
    return nullptr;
}
```

One key area is minimizing the overhead of function calls on critical code paths. A function call generally involves setting up a new stack frame, passing parameters, saving register state, and restoring it on return. If the number of iterations is high enough, that overhead meaningfully hurts performance. This shows up especially clearly when functions do simple work whose runtime is comparable to the cost of the call itself:

```

struct Vector3D { // Unoptimized code
    double x, y, z;
    double getX() const { return x; }
    double getY() const { return y; }
    double getZ() const { return z; }
    double magnitude() const { return sqrt(x*x + y*y +
z*z); }
};
// Slow version with multiple function calls
double calculateDistance(const Vector3D& v1, const
Vector3D& v2) {
    double sum = 0.0;
    for (int i = 0; i < 1000000; ++i) {
        double dx = v1.getX() - v2.getX();
        double dy = v1.getY() - v2.getY();
        double dz = v1.getZ() - v2.getZ();
        sum += sqrt(dx*dx + dy*dy + dz*dz);
    }
    return sum;
}

```

Compiler developers know about this, and there are various optimization techniques to deal with it. Inlining lets the compiler embed a function's code directly at the call site, eliminating the overhead. Hoisting invariant computations out of loops (loop hoisting) prevents the same operations from running over and over. Merging several simple functions into one cuts down on branching and improves data locality in the processor cache. Sure, the compiler will probably spot these simple getter functions and manage to optimize that part, but it can't write a fundamentally better solution for you:

```

struct OptimizedVector3D {
    double x, y, z;
    inline double getX() const { return x; }
    inline double getY() const { return y; }
    inline double getZ() const { return z; }
    // An optimized distance function
    inline double distanceTo(const OptimizedVector3D&
other) const {
        const double dx = x - other.x;
        const double dy = y - other.y;
        const double dz = z - other.z;
        return sqrt(dx*dx + dy*dy + dz*dz);
    }
};

```

The alternative with `OptimizedVector3D` relies on inlining the getters, which lets the compiler embed their code right at the call site, plus a new function, `distanceTo()`, that does all the math locally without extra calls. Pay special attention to memory allocations inside loops or hot functions. Dynamic memory allocation is one of the most expensive operations around, especially when it happens often in heavily loaded code paths. Methods for optimizing code like this are covered in more detail in *chapter 0x25, "Not-So-Boring Programming."* Source-level and function-level optimizations have varying degrees of impact on performance depending on the hardware platform. On high-performance systems (modern game consoles and desktop PCs with multi-core processors, large L2/L3 caches, and high-frequency memory), local code improvements rarely yield more than a 2-3% gain and don't noticeably move the needle overall (though you should never discount human error). Improvements like that rarely justify the developer-hours spent on them from a business-metrics standpoint, since developer time is considered a more valuable resource for building new functionality or fixing critical bugs. On handheld devices, though, mobile ARM processors with limited clock speeds (1-3 GHz versus 4-5 GHz on PC), small caches (512 KB-2 MB L2 versus 8-32 MB), strict power-consumption limits, and thermal constraints mean the exact same optimizations can deliver a noticeable gain of several dozen percent, and meaningfully affect frame-rate stability and battery life.

```

long unpredictable_branches(const vector<int>& data) {
    long sum = 0;
    for (int val : data) {
        if (val > 0) // the data is random, the branch is
unpredictable
            sum += val;
    }
    return sum;
}

// Avoiding branches via bitwise operations
long branchless_version(const vector<int>& data) {
    long sum = 0;
    for (int val : data)
        sum += val & (-(val > 0)); // Use a mask instead
of a branch
    return sum;
}

```

Finding and applying effective (micro)optimizations is its own area of development, one that requires a deep understanding of processor architecture and the memory system. Writing a few lines of truly optimal code can take days of analysis: detailed profiling to find bottlenecks, cache-behavior analysis, studying the assembly, and understanding branch-prediction quirks on a specific architecture. How effective these optimizations are depends heavily on context: memory access patterns, how big the working data set is relative to the different cache levels, data alignment to cache-line boundaries, how often a given piece of code runs, and how it interacts with the rest of the system. Many developers treat this as a last resort, preferring changes at other architectural layers, and in most cases that's the right call.

Game Replays (~time?)

Game replays are one of the most powerful debugging tools out there, especially for complex systems with lots of interacting objects and nonlinear logic. A replay isn't just a video recording; it's a time-ordered set of inputs (game commands, events, states) that the engine plays back from scratch. This lets you reproduce the game's behavior frame-perfectly, regardless of the machine's performance or time of day, which makes it ideal for deterministic debugging. It's effectively the equivalent of time-travel debugging, but without needing to hook into the compiler, without

"freezing" the state of an entire virtual machine, and with minimal overhead. Technically, a software replay requires recording action points for objects on every iteration of the game loop: player commands, AI states, network messages, random-number-generator changes, and so on. With the right architecture, a replay stays reproducible even months later. This lets QA or a developer reproduce a bug just by "scrubbing" the match to the right moment. Replays themselves can also be repurposed for writing automated tests, simulating game scenarios, and even training AI. In production, it often turns out to be more practical and powerful than classic debuggers or logs.

Note

A developer's time is the most expensive resource there is, and investing it in systems like replays pays for itself many times over in the hours saved on debugging and testing.

Component Isolation (~5%)

Modular component isolation is an architectural principle built around separating interfaces from implementations and using dependency injection. Properly designed modules interact only through defined API boundaries, hiding internal implementation details and keeping subsystems loosely coupled. An audio system might expose an interface for sounds through methods like `PlaySound()` and `SetVolume()`, while hiding whether it's actually using `DirectSound`, `OpenAL`, or a custom audio pipeline under the hood. This kind of isolation lets you swap implementations without touching the client code. An isolated architecture makes porting and platform-specific optimization easier, since each component can be adapted independently of the rest of the system. A renderer can have separate implementations for `DirectX`, `Vulkan`, `Metal`, or `OpenGL`, selected at init time, while the game logic stays untouched. Likewise, the input system can use different implementations for keyboard/mouse, gamepad, touch input, or VR controllers, all behind one unified interface for the game code. All of this lets you build platform-specific solutions without merge conflicts, test individual components' performance in isolation, and roll out changes without risking the stability of the whole system.

Specific Low-Level Optimizations (<3%)

There's a common misconception that optimization at this level requires complex, hard-to-understand solutions, like writing assembly, applying advanced and intricate algorithms, or converting everything wholesale to SIMD. The good news is that in practice, that's not always true, and plenty of

optimizations at this level turn out to be fairly simple and intuitive. Often, clean and readable code is already fast on its own, simply because it's easier for the compiler and optimizer to work with. But not always.

Everything Else (~10%)

What's there to say about this part? It usually covers things like sane build pipelines, where a build reaches QA an hour after a commit and comes back to the author within two if there were errors. Also worth remembering: good debugging tools and replays, and modern game-dev tooling (BT: behavior trees, VS: visual scripts, blueprints, AS: animation scripts). Debugging through a debugger is often painfully hard, and bugs tied to these systems fail to reproduce 99% of the time, though that admittedly depends a lot on how skilled your QA team is. These improvements might not have a direct effect on the game's runtime performance, but they help the team find and fix problems faster, and when a developer builds a feature instead of fixing a bug, well, they've "done some good" for the project, in their own way.

Benchmarks

Let's talk about benchmarks, the cornerstone of any approach to optimization. You can't improve what you can't measure. A good benchmark is a compass in the ocean of optimization, giving you immediate feedback on whether a given change actually worked. In essence, a benchmark is an execution scenario that reproduces a specific load condition and serves as a baseline for quantitatively comparing the performance of different versions of code. For that reason, a benchmark should meet four critical requirements. "*Reproducibility*" is a fundamental property that often gets overlooked, but running the same benchmark repeatedly on an identical setup should give you statistically consistent results (minor, bounded deviations are fine). This property depends on fixed input data, deterministic behavior for "random constants" via a fixed seed, stable system conditions, and maximum isolation from external factors like network activity or background processes.

Speed (speed test) is a practical requirement for efficient development. A good benchmark rarely takes longer than a few dozen seconds, letting you iterate on optimization without long pauses.

Representativeness/realism (representativeness, realistic) comes from modeling real game scenarios with usage patterns similar to the actual CPU, GPU, memory, and I/O load. The test should reproduce typical game workloads: physics simulations, rendering, or peak AI-computation moments. Avoid purely synthetic tests that aren't tied to the engine's real architecture. *Sensitivity to change* (bench sensitivity) guarantees that the benchmark can detect both improvements and regressions in performance.

Statistical samples shouldn't be based on fewer than 10-20 runs. A good benchmark should show a noticeable difference (5-10%) for meaningful code changes while staying stable for cosmetic edits that don't affect performance. For instance, you might notice the frame rate drops in certain game scenes. Before starting optimization, you need to lock down a reproducible scenario: pick a specific spot in the game where the problem shows up, and make sure you can get to it quickly. That lets you take repeated measurements under identical conditions, though in practice that's not always possible, which is why teams build a dedicated benchmark scene or mode with strictly deterministic behavior, so every run gives similar results and the measurements become more objective.

The whole point of benchmarks is that they let you quantitatively confirm the comparison between the original and optimized states. If you can, always start with measurements, since that's how you can quantitatively confirm an improvement. Despite how simple it sounds, building good benchmarks is full of pitfalls. One of the most common mistakes is measuring "warm," ignoring the first runs because of warm-up or cache-filling effects, while forgetting that in a real game, "cold starts" and "cold data" show up just as often as "warm" ones. Automating benchmarks through CI/CD lets you catch performance regressions early, and visualizing the results as graphs helps you spot long-term trends and correlations between different performance metrics.

Measurement

The next step in optimization is gathering and analyzing performance metrics that let you pin down the system's bottlenecks precisely. Every change should produce a measurable difference in performance for a minimal investment of resources and time, in other words, deliver a good return on invested effort (ROI). That's not always a positive result, but it still helps you figure out which direction to take the next step. Proper diagnosis confirms that your changes are targeting a real problem, not some random part of the code that doesn't actually affect efficiency. Metrics, profiles, and logs all serve as sources of information, letting you avoid "blind" changes and unnecessary iteration. The approach to optimization work is similar to engineering analysis: first assess the system at a macro level, then gradually narrow in on specific cases. At the top level, it's important to figure out which component is limiting performance: CPU, GPU, or memory. In most cases that actually need intervention, the bottleneck turns out to be tied to a specific module or piece of code (this is the algorithms-and-data-structures level). Profilers, system monitors, tracing tools, and the engine's own performance metrics all help with this diagnosis. They let you identify "hot spots": functions or routines (this is the source-code level) eating up a large

chunk of CPU time, GPU time, or memory. When analyzing, it's worth starting by checking CPU and GPU load with general-purpose monitors, and if the GPU isn't overloaded, a profiler can help measure how much CPU time goes into updating and processing the game's logic. Finding problems at the top level lets you narrow the scope of your analysis and focus your optimization in one place. But when measuring, it's important to understand that "one problem" is rarely a single, isolated point: a game is made up of countless micro-operations, and bottlenecks often turn out to be a combination of several factors. At the instruction level, that might be a long call that stalls the processor; at the graphics-pipeline level, it might be unbalanced rendering-command processing. This can look like two different problems, but fixing one, you'll notice changes in the other. Using different levels of analysis, from micro to macro, lets you identify the single component holding back performance the most and reallocate resources effectively. It's important to always start by analyzing the overall performance picture and gradually drill down to specific bottlenecks, combining tools and metrics to precisely identify and fix the real problems, rather than jumping at the first thing you see. It's better to make a list of candidate spots; it often gives you a broader picture that points you in the right direction. Finally, smart optimization has to be paired with an assessment of implementation cost; sometimes fixing a specific bottleneck takes a huge amount of effort for a minimal payoff, and in cases like that, it makes more sense to shift your attention to more productive areas. The "effect-to-cost" ratio can become yet another optimization parameter in its own right, helping you avoid getting carried away with micro-optimization. The end result is that optimization stops being a pile of random improvements and becomes a managed process that improves the efficiency of the whole system. All of this calls for a systematic, step-by-step approach, with an assessment of overall performance, identification of critical bottlenecks, and a balance of costs.

Changes and Engineering Intuition

Once you've gathered enough data and identified the bottleneck limiting performance, it's worth setting those results aside and running the analysis again from scratch, deliberately avoiding that very bottleneck. This technique is known as "hotspot avoidance": temporarily excluding or ignoring the system's main bottleneck while analyzing performance. The point of doing this (and in 80% of cases, we really don't find any other major slowdown spots) is to stop focusing on the already-identified "hot" code path for a moment, and instead investigate how the rest of the system's components behave. But this step lets you uncover *hidden* bottlenecks that might otherwise go unnoticed while the obvious "hot spot" dominates the profiler,

giving you a fuller picture of how the load is distributed across components. The real value of this method shows up later, once you've built up a lot of experience optimizing different systems. Setting the main bottleneck aside lets you spot points that, together, can deliver a meaningful performance gain, or that influence the "hotspot" itself. It also makes it easier to analyze complex systems that create unexpected interdependencies (engineering intuition). Now you can start making changes; they might involve modifying algorithms, optimizing data structures, changing memory access patterns, or "rewriting everything from scratch." But even experienced optimizers often have to test several hypotheses (and changes) about the best way to improve performance, so it's worth trying out a few different implementations before settling on a final solution, and that's exactly where the ideas you gathered during the "hotspot avoidance" step come in handy. After making changes, we measure performance again with benchmarks and profilers, because there's no way to tell whether fixing the bottleneck actually helped, or whether the problem was really somewhere else. Systematic measurement doesn't just reveal the real causes of bottlenecks; it builds up experience and develops "engineering intuition," letting you solve similar problems faster and more precisely on future projects. My colleagues with a lot of engine experience would often pin down a problem just by reading logs and a general profiler snapshot, without even looking at the code itself, and in most cases, their recommendations for changing one part of the system or another turned out to be right, and that's what I'm getting at with "engineering intuition." Engineering intuition grows out of studying what's hidden behind the raw measurements, letting you see the system as a whole instead of fixating on a single point.

Hot Functions and Bottlenecks

A key task in the optimization process is identifying "hot functions" (code paths with heavy computational load) and "bottlenecks" (components that limit overall throughput). The problems can be split into three categories: abstractions, smoothness, and hardware limits. Abstractions, or more precisely, premature abstractions that create unnecessary layers in the engine's architecture and overcomplicate the system, include things like multiple inheritance, one-size-fits-all solutions for game-dev problems, and overblown design patterns where plain procedural code would do just fine. This also includes writing "pretty" code instead of "fast" code, where performance-critical sections get weighed down with unnecessary niceties and checks ("keep it simple, and people will come around"). Smoothness, as most developers point out, matters more than peak performance: an uneven frame rate from sporadic memory allocations on the main thread, inconsistent frame times from algorithms with variable complexity, "stutters"

from blocking resource loads. The point is that a steady 30 frames per second beats bouncing between 60 and 20: the player experiences it visually, and a consistent fps is a clean, understandable quality metric. Hardware limits are "creative constraints." Trying to brute-force your way past a platform's physical limits instead of optimizing smartly just adds complexity with no real benefit to the system. More polygons without a matching level-of-detail system reads to the player as low fps, complex shaders that ignore fill-rate limits read as low fps, memory-hungry data structures that ignore the cache hierarchy read as low fps, and so on. Modern CPU-GPU architecture is a complex data-processing pipeline with multiple parallel execution units. In this "digital hell," data flows through a hierarchy of caches, moves between the CPU and GPU over a bus, and gets synchronized between threads via memory barriers and atomic operations. The system doesn't do exactly what you wrote in your code. Stable, correct operation comes from balancing the load across all these components, but even then, you won't manage to push CPU utilization past 70-80%, though you can do a better job of keeping the GPU busy and using memory bandwidth and caches more efficiently. And you have to find a delicate balance right at the edge of what's possible among all of this, or you'll end up with resource contention, where one component becomes the critical link, causing pipeline stalls, context-switch overhead, and reduced throughput across the rest of the system.

Hot Functions (Hotspots)

In software systems, "hot" functions are localized sections of code with abnormally high resource consumption (CPU, memory, disk operations). They're marked by a disproportionately high execution frequency, heavy CPU utilization, heavy memory pressure, or excessive system-call overhead. The Pareto principle (the 80-20 rule), applied to code optimization, means that 20% of the source code accounts for 80% of the program's total runtime, which makes targeting "hot" spots a highly effective strategy. Iteratively hunting down these spots lets you find the code paths consuming the most resources and apply targeted optimizations to lower their "temperature." Once you fix one problem, the load shifts, and previously less significant spots become the new "hot" spots that need attention. This process continues until you hit acceptable results or run out of time to spend on it. It's worth noting that an engineer's time is itself a limited resource, in the sense that the more time a developer spends on changes in one piece of code, the fewer changes they'll have time to make elsewhere. Modern profilers give you detailed, instruction-level analytics, letting you identify not just temporal "hot" spots, but also sections with a high number of cache misses, heavy I/O operations, and lock contention in multithreaded applications.

Bottlenecks

Picture a cook preparing a dish at a café. The cook can't do the whole dish alone, so they delegate preparing the side dish to an assistant while they handle the main part themselves; then the two components come together and the waiter carries off the finished plate. What's the total prep time going to be? Say the cook finishes their part, but the assistant can't work any faster: they're the one slowing down the whole system. Games work on exactly the same logic: there's always a "slow assistant," a bottleneck: the CPU, the GPU, memory, or something in the code holding back the overall pace. Our job is to find that weak link and try to get it working as well as everything else. Optimization is almost always about finding trade-offs; 99% of the time, you won't have a silver bullet to solve the problem in one stroke. The first and most common trade-off is speed versus memory; sometimes it's cheaper to store extra data than to recompute it from scratch. This is called "memoization," or caching: compute something once, store it, and then just pull the result from memory. Yes, it costs more memory, but you get the result faster. The second trade-off is accuracy for speed. Sometimes you can "cheat" and speed up a system by simplifying a data type or using a less precise algorithm. Shaders have log and logp functions: the first is accurate but slow, the second is fast but less precise. In the final image, you probably won't notice if a neighboring polygon lands a quarter of a pixel to the right, but the game will run noticeably faster. This principle applies to almost any data model, pathfinding, or game AI; the main question is how much you're willing to "cheat." In more formal terms, this is called reallocating resources from low-priority computations toward higher-priority ones. And the third classic trade-off is simplicity versus speed (this one's more about people and the team). The classic case: a young, sharp developer discovers templates and caching with five levels of nesting while trying to optimize the texture-loading system. I can say with a fair amount of confidence that they'll write that system, and it might even run faster, using intrusive pointers, custom allocator pools, and a lock-free hash table. The result is predictable: code that nobody, including the author themselves a month or two later, can understand, debug, or modify without risking breaking it. Other developers on the team start avoiding those files, cry over changes to that system, and code reviews regularly fill up with comments like "what in the world is this, and can we rewrite it in plain C++?" An experienced developer knows that real performance gains don't come from showing off algorithmic chops, but from understanding the trade-offs between maintainability and the payoff of an optimization. Sure, you can squeeze out an extra 5% of performance with custom memory pools, but that'll cost you two weeks to implement and another month debugging weird edge cases. Maybe it's worth just

implementing a linear allocator and trading memory for time instead? Mature optimization means fixing the truly critical spots with clean, well-documented solutions that can easily be switched off via conditional compilation if problems come up. Because in reality, shipping the game matters more than elegant algorithms, and code the team can actually maintain matters more than code that looks impressive at a conference.

Misconceptions and Myths

By its nature, software optimization never truly ends: there's always something to improve, but past a certain point you're only getting diminishing returns, where more effort produces less and less noticeable gains. What's more, developers tend to spend time chasing directions that don't actually help (though they make for great conference talks), while just adding to the system's maintenance burden. The key skill here is knowing when to stop and being able to judge where improvements will actually pay off versus where they're turning into a "witch hunt." One of the most common mistakes is working off assumptions. You can't "guess" what's slowing a system down, because guesses like that are usually wrong. The only reliable path is measurement and analysis, analysis and more measurement, turning guesses and subjective impressions into solid engineering decisions. Optimization isn't limited to the post-hoc stage, after the code is written, since 80% of performance problems get resolved back at the architecture-design stage. Smart decomposition, a clear object hierarchy, and well-thought-out dependency management help you avoid bottlenecks in the first place. Even accounting for the overhead of languages like C# or Java, a well-structured system often outperforms code written in "pure C++" with no regard for architectural constraints.

When optimizing, it's important to focus on worst-case scenarios, not average ones. Most developers test their game or app on powerful dev machines, which often paints a distorted picture; "weak" hardware exposes completely different bottlenecks: instead of the GPU, the app might be hitting a memory-bandwidth ceiling or a weak CPU. To properly diagnose optimization problems, you need to deliberately test on minimal or low-end configurations, since those are what actually show behavior close to what a chunk of your real audience experiences. Players "notice" the difference between 55 and 60 fps, though they rarely complain about it, but drops to 20 fps (even rare ones) read as serious lag, which is why consistent frame time often matters more than the absolute average, and comparing debug and release builds gives fundamentally different results. Debug builds have compiler optimizations turned off, include extra checks, and can run several times slower, and unevenly so. For valid performance profiling, you need to

work only with release builds, including debug symbols where needed or disabling optimization for just a few specific spots, which lets you balance traceability against measurement with a realistic view of speed.

A debug build, though, is useful early in testing, since it can catch excess CPU usage or suboptimal algorithms before they become critical.

Finally, a key requirement for benchmarks is controlling your external variables. A poorly run test can create the illusion of an optimization: say, the first measurement is taken in fullscreen mode, and the second in windowed mode, at a different refresh rate or resolution. The frame rate ends up rising because of the changed environment, not the code, and the comparison becomes meaningless. To rule out mistakes like this, you need to lock down every environment parameter and change only one variable between runs: the piece of code you're optimizing.

The best practice is automating your benchmarks, which removes human error and guarantees reproducible results. A lot of people think optimization time should only get added to the sprint once the game is nearly finished, or worse, once problems actually show up.

But if you ignore optimization early on, you risk running into increasingly complicated development and unpredictable performance issues at the worst possible time.

Another myth: reactive optimization is effective and sufficient. Sure, hundreds of games ship having been optimized "after the fact," but that approach usually ends up cutting features or simplifying mechanics just to compensate for low fps. Reactive optimization creates a false sense of control, since fixing problems late in development is far more expensive and labor-intensive than doing systematic work throughout the whole project.

The next myth has to do with needing to carve out "big blocks of time" for optimization. In practice, it's far more effective to work it in gradually; approaches like a "performance budget" let you set limits for FPS, CPU, and GPU usage ahead of time, and build the team's work around those targets. Constant monitoring and adjustment beat rare last-minute pushes in the final weeks of development.

One more myth worth remembering: every optimization is worthwhile. In reality, there are plenty of "black holes" that eat up programmer time without any real payoff. You should only optimize what actually affects performance, weighing the potential payoff against the time spent making the change. This mindset helps you avoid wasted effort and focus on the bottlenecks that genuinely slow the game down.

Finally, the myth that performance is a single metric. In reality, a game's "slowness" has multiple dimensions: FPS, response time, CPU and GPU load, and frame-rate stability. To effectively optimize even one of these, you first

need to understand which ones actually matter for your particular project, how they relate to the others, and what's really getting in the way.

Only this kind of systematic approach lets you sidestep the common traps around optimization. A separate question is whether changes like this will actually get approved. Probably not: the odds that the project's architect, tech lead, or technical director will sign off on changing the architecture sit at about 99.9999%. That's not a programmer's call to make, and decisions like that are essentially poaching the architect's job, not to mention that changing an existing, working, battle-tested system risks breaking its neighbors, and when that bug surfaces (and it will), nobody can say when. That last reason is exactly what keeps a developer who started yesterday from rewriting half the engine to do it "the right way," leaving them to make do with changes at the code and algorithm level, which can still "do some good" to the tune of 10%+ in performance. And that brings us to the reality that game development forces us to balance our "ideal" technical solutions against a project's practical constraints.

Even once you understand every myth about optimization, a programmer is often still forced to work within a limited, existing architecture, settling for local improvements at just the algorithm and code level. But even that approach can deliver real results: that same 10-15% performance gain that can make the difference for a successful launch. The main thing is understanding the limits of your influence, picking the right places to apply effort, and remembering that optimization isn't just a technical skill, it's also the ability to work within a team, where every change needs to be justified, measurable, and agreed upon. In the end, the best optimization is the one players actually notice, not the one you get to brag about in the tech chat.

Tools

Any work on improving a program's performance is pointless if you can't see what's actually happening inside your code. It's like trying to fix a watch blindfolded, or tune a musical instrument with your ears plugged: you can turn screws and pluck strings, but the result will be random. In everyday life, we get feedback instantly: trip, and you fall; miss a turn, and you hit a dead end; overdo the salt, and the soup's too salty. These signals let us fix the mistake right away and avoid repeating it, but the world of programming works differently: code can run millions of times a second, process gigabytes of data, and interact with dozens of system components, while all we see is the final result, say, that the fps won't climb above fifty. Modern programs and games, games especially, contain hundreds of thousands of lines of code running on top of complex operating systems, GPU drivers, and various libraries that have even more code inside them. Without specialized analysis

tools, all we can do is guess at what's happening where in the problem code. Maybe the CPU can't keep up with the computations, maybe the bottleneck is loading textures off disk, or maybe memory is running out because of leaks. Profilers, performance monitors, and debuggers are tools that can also mislead you, but you can't do without them. They show which functions run longer than others, how much memory each part of the program consumes, how often disk access happens, and where the CPU sits idle waiting for data. Without this data, we're completely blind; once we have it, the solution often becomes obvious, but knowing how to measure and analyze performance correctly is a tricky science, one that sometimes borders on alchemy. But any tool that measures a program's performance inevitably affects that performance itself: some tools run cleanly and barely slow the program down at all (these are called non-intrusive), while others can meaningfully reduce the application's speed (these are considered intrusive). How much a tool interferes depends on how much data you need and how often. A simple FPS counter has almost no effect on the game, while a detailed analyzer that logs every function call and its runtime can slow the program down several times over. Some tools, like Valgrind, go even further and run the entire program inside a virtual machine to get the most detailed information possible about every single operation; you can imagine what that does to performance, the program can end up running an order of magnitude slower than usual. There are a few distinct approaches to measurement. The gentlest and simplest is sampling profilers, which periodically "photograph" the program to see which code is running at that instant. By taking a certain number of these "snapshots," you can build up a statistical picture of where the program spends most of its time. A step heavier are interceptor tools, which sit between the program and external resources, like PIX, which intercepts every call to the GPU through the DirectX API, or network analyzers that "eavesdrop" on all internet traffic. To get even more detailed information, you have to modify the program's source code, adding special profiling commands and markers to collect data, a process called instrumentation. Which tool to pick depends on your specific goals and how willing you are to trade performance for depth of analysis. The main rule: the tool shouldn't distort your measurements more than the problem you're actually looking for. There's no point optimizing a function that takes up 1% of runtime if the profiler itself is slowing the program down by 50%, which is why teams often use all three kinds of tools, just wired in under different conditions.

Theory and Terminology

Profilers are the general name for performance-analysis tools used across most areas of game and software development. A profiler's core job is to gather the most detailed information possible (at whatever level the developer sets) about how load is distributed at runtime.

The classic scenario is measuring CPU time spent, but modern profilers can track a lot more: cache misses, memory allocation volume, disk access, thread usage, and synchronization. Depending on the instrumentation level, you can either get an aggregated picture (which modules or systems are loading the CPU the most) or see a frame-by-frame or interval-based slice, a call stack, and the exact load distribution down to individual instructions.

In practice, profilers split into general-purpose and specialized ones: the general-purpose kind are usually built into an IDE or a platform's SDK (Visual Studio Profiler, say) and give you a rough sense of where performance is being lost. The specialized kind are low-level tools (Sony Razor, Intel VTune, PIX, or Tracy, for instance) that gather data at the level of CPU counters and OS events, letting you analyze your code's behavior far more precisely. Only low-level tooling lets you measure how a specific algorithmic step or a chosen memory-usage scheme actually affects that section of code's final performance.

Total time

Total time shows a function's cumulative execution time, including every nested call to other functions. In theory, it gives you the full picture of how much time the program spends on a given piece of code, accounting for all its dependencies, and it's useful for finding "hot spots" in an application, but in practice it can paint a distorted performance picture. With inline functions and compiler optimizations, where the compiler embeds them directly into the calling function's code, you get a "blurring" effect on the parent function's self time, even though the inlined code is actually doing the work. This metric also doesn't account for indirect costs like cache pollution or data eviction from memory, which can meaningfully affect how trustworthy the data is. For instance, when analyzing functions that are heavily input-dependent (sorting an almost-sorted array versus an unsorted one), the profiler will show an averaged value, which can mask real performance problems in specific usage scenarios.

Self time

Self time measures only the time spent executing instructions inside the function itself, excluding the runtime of every function it calls. This helps you find code blocks that are actually doing the expensive work, rather than

just delegating it to other functions; functions with complex algorithms, loops, or "number crunching" usually have high self time. The function *ObjectUpdate()* might show a high total time, since it calls every other game-state-update subsystem, but a low self time, since it does very little work on its own. Meanwhile, a collision-physics function might show significant self time, pointing straight at the spot that needs optimizing. The problems here are nearly the same as with the previous metric: self time can't reflect a function's real impact on the system's overall performance; a function might show low self time while still triggering a pile of expensive operations through other functions, creating a false impression of efficiency. On top of that, the metric doesn't account for function-call overhead: if a function makes thousands of tiny calls, the overhead of setting up stack frames, passing parameters, and returning values can be significant, sometimes close to the actual runtime, while staying completely "invisible" in the self-time number.

Another problem comes from "aggressive optimization." This is especially common with clang-family compilers, when they aggressively apply inlining, or partially embed small functions' code directly at the call site. As a result, self-time numbers get skewed, since code that logically belongs to a separate function ends up counted as the calling function's self time.

Call count shows how often a given piece of code runs during a profiling session, helping you understand how different functions get used across the application. Functions with a high call count often become optimization candidates, even if their individual runtime is small, since the combined overhead of many calls can add up to something significant. Looking at call counts is usually the first step in spotting inefficient algorithms or redundant computation (with one caveat, see the note below).

If a function gets called thousands of times per frame, that can point to a need for various optimization techniques; a high call count can also point to recursion, resource leaks, or other problems in the application's architecture.

Overall, this is a pretty important metric, but the call count alone doesn't tell you how those calls are distributed over time: a function could easily get called 100,000 times in a session, but if all those calls happen during level loading, they have no effect on gameplay performance. The profiler will show a high call count here without indicating that the function is inactive during frame rendering. Or the call count can be misleading about real architectural problems: a high call count might be the result of a well-thought-out design with lots of small, specialized functions, rather than a sign of inefficiency.

Note

Modern compilers have gotten quite good at inlining, so focusing exclusively on call counts can lead to premature and incorrect "conclusions."

Total wait time is the time a function's thread spent in a blocked state, waiting for resources to free up, I/O operations to finish, or system events.

This time has nothing to do with active execution; it reflects delays caused by external factors. High wait time can point to indirect problems elsewhere that fed into the function's total time. This metric usually comes up during refactoring, or when trying to reduce dependencies between functions.

The Story of One Optimization

The game levels I got to work on for mobile had an internal format similar to JSON that required escaping certain characters. Most of the development happened on PC, where there were no performance problems at all. The game could receive external changes in several formats from different servers (JSON, bson, XML, kvpatch), and on top of that, players could create and share their own levels, district layouts, and individual buildings (custom YAML). All of this data moved through the game's servers, which sent changes and updates in binary form (to save on bandwidth), and to process them correctly, that data had to be converted back to text, normalized into JSON, and then converted again into the internal format so the level parser could actually read it. Levels were requested fresh from the server every time, so players couldn't "cheat themselves some cash," since part of the simulation ran locally on the device. Data could arrive in different formats from different systems, and over the course of processing, it got fully checked for escaping needs 4 (four!) separate times. My point here is that an architectural change would have let us strip out all the unnecessary steps described below, but getting approval to rewrite server code just to "win back" two seconds of level-load time for the end user wasn't going to happen (read: basically never). This was, in its purest form, an architectural mistake that the system architect didn't catch in time, and it leaked down into the algorithm-and-code level. Games in general are full of text data: it's in configs, in localization, in resource names. And strings are often wrapped in quote characters ". But what happens when the string itself contains a quote character? In that case, the string needs escaping. For example, the quote character " or the backslash \ needs to be replaced with or . You could easily write code in your favorite language to check for this. Most strings don't need escaping at all. So it's useful to have a fast way to check whether a string needs escaping. The algorithm is dead simple: a function takes an argument of type `eastl::stringz` named `v` (`string_view` didn't exist yet in the C++14 standard, but custom solutions for non-zero-terminated strings already did). The function walks every character in the input string `v`. Inside the loop, it first uses the comparison `(uint8_t(c) < 32)`, which checks whether the character's ASCII value is less than 32. This condition was there to catch control characters (newline, tab, and so on). The format had some differences from JSON, so these characters also needed escaping for the parser to work correctly.

```

bool need_escaping_symbol(eastl::stringz v) {
    // stringz -> string_view: a lightweight reference to
    the string
    // with no ownership of the memory
    // Walk every character of the string

    for (char c : v) {
        // Check three conditions with bitwise OR (|)
        // instead of logical OR (||) for speed:
        // this avoids short-circuit evaluation and branch
        prediction
        if ((uint8_t(c) < 32) // Control characters (0-
31): \n, \t, \r
            | (c == '"')      // Double quote: must be
escaped
            | (c == '\\')     // Backslash: the escape
character itself
        {
            return true;
        }
    }
    return false;
}

```

Then the comparison (`c == '"'`) checks whether the character is a double quote, and `c == '\'` checks whether it's a backslash. If even one of these conditions is true for any character, the function returns true, indicating that the string needs escaping. If none of the conditions hold for any character, the function returns false, meaning the string doesn't need escaping. Simple? Well, not complicated, at least, and understandable even to a beginner. On PC (in simulation), this code ran plenty fast and needed no changes at all. On mobile, though, things were grim: an empty level took more than 10 seconds to load, 2 seconds of which went into parsing and applying the text configs. And as the number of objects on a level grew, that time kept climbing, until a fully populated level loaded in more than 2 minutes on the top-of-the-line iPhone 5s at the time, which the QA team and users were understandably not thrilled about. Fifteen seconds of load time was the absolute max we could count on to pass Apple Store certification.

	iPhone 5S	iPhone 6S	Simulator	PC
Empty Level Time (parsing time)	12s (3s)	10s (2s)	3s (0.152s)	2s (0.112s)
Full Level Time (parsing time)	143s (23s)	92s (16s)	34s (1s)	22s (1.199s)

One thing worth noting: this function stops as soon as it finds a character that needs escaping. The problem with this approach is that the compiler can't vectorize these computations because of that internal `if`. And that conditional early exit actually ends up being more expensive than just scanning the whole string with no conditions at all.

```
bool linear_need_escaping_symbol(eastl::stringz v) {
    bool b = false;

    for (char c : v) {
        b |= ((uint8_t(c) < 32) | (c == '\'' ) | (c == '\\'));
    }
    return b;
}
```

Getting a bit of extra performance just by dropping the *if* is nice, but clearly not enough to make the QA team's nagging reports go away. What we can do, though, is think harder and get rid of the arithmetic and comparisons in the code entirely. To do that, we need to build a lookup table that already contains the check result for every possible byte value, a flag for whether it needs escaping or not. That takes a bit more effort, but the result turns out pretty fast: checking each character only takes one table load:

	iPhone 5S	iPhone 6S	PC
Naive	0.055 GB/s	0.075 Gb/s	0.650 Gb/s
No branches	0.092 Gb/s	0.112 Gb/s	0.900 Gb/s

```
constexpr eastl::sarray<uint8_t, 255>

vflt_quotable_character = []() constexpr {
    eastl::sarray<uint8_t, 255> result = {0};
    for (int i = 0; i < 32; i++) {
        result[i] = 1;
    }
    result[''] = 1;
    result['\\'] = 1;
    return result;
} ();
```

```
bool table_need_escaping_symbol(eastl::stringz view) {
    uint8_t needs = 0;
    for (uint8_t c : view) {
        needs |= vflt_quotable_character[c];
    }
    return needs;
}
```

	iPhone5S	iPhone 6S	PC
Naive	0.055 GB/s	0.075 Gb/s	0.650Gb/s
No branches	0.092 Gb/s	0.112 Gb/s	0.900Gb/s
Table	0.215 Gb/s	0.255 Gb/s	1.700Gb/s

Can we go faster still? Yes, if we reach for SIMD instructions: NEON on mobile, or SSE on PC. And since strings are usually not very long (60% of strings are under 64 characters: keys, resource names, flags, and just plain short strings), the general idea is to load data in 16-byte chunks and run comparisons across those bytes all at once. There's no magic here: why compare one character at a time when you can run N comparisons simultaneously? It should simply, physically, run faster. And what do we do if we've got fewer than N characters left, or a "tail"? Well, then we just fall back to the solution we already have, which already performs reasonably well. For SSE, the solution looks roughly like this (NEON would be similar):

```

inline bool simd_need_escaping_symbol(eastl::stringz view)
{
    int size = view.size();
    if (size < 16) { return
linear_need_escaping_symbol(view); }
    size_t i = 0; __m128i r = _mm_setzero_si128();
    for (; i + 15 < size; i += 16) {
        __m128i word = _mm_loadu_si128((const __m128i*)
(view.data() + i));
        r = _mm_or_si128(r, _mm_cmpeq_epi8(word,
_mm_set1_epi8(34)));
        r = _mm_or_si128(r, _mm_cmpeq_epi8(word,
_mm_set1_epi8(92)));
        r = _mm_or_si128(r, _mm_cmpeq_epi8(_mm_subs_epu8(word,
_mm_set1_epi8(31)),
                                _mm_setzero_si128()));
    }
    if (i < size) {
        auto word = _mm_loadu_si128((const __m128i*)
(view.data()+view.length()-16));
        r = _mm_or_si128(r, _mm_cmpeq_epi8(word,
_mm_set1_epi8(34)));
        r = _mm_or_si128(r, _mm_cmpeq_epi8(word,
_mm_set1_epi8(92)));
        r = _mm_or_si128(r, _mm_cmpeq_epi8(_mm_subs_epu8(word,
_mm_set1_epi8(31)),
                                _mm_setzero_si128()));
    }
    return _mm_movemask_epi8(r) != 0;
}

```

It turned out pretty well. We closed out this task with a clean conscience, 2016 was just around the corner, and we went off to celebrate our little victories at the company party. The render team, meanwhile, kept right on packing elephants and optimizing something or other, but that's a whole different story.

	iPhone 5S	PC	iPhone 6S
Naive	0.055 GB/s	0.650 Gb/s	0.075 Gb/s
No branches	0.092 Gb/s	0.900 Gb/s	0.112 Gb/s
Table	0.215 Gb/s	1.700 Gb/s	0.255 Gb/s
SIMD	0.480 Gb/s	4.200 Gb/s	0.580 Gb/s

Packing Booleans into Bits

Games often use the pattern of packing boolean values into bits. This is handy for optimizing memory and speeding up bulk checks. For example, checks like these might include whether the player is inside some area of the map, whether cells on a four- or six-sided grid are walkable, or other spatial checks that need to run fast. It's not rocket science, but when a profiler flags one of these functions as "hot," you might get curious about exactly how it works and whether it can be optimized. A bitset data structure is a way to efficiently represent a set of integer indices, while also supporting various operations on them, like union, difference, and intersection. So, every unit might occupy one or several tiles, especially if it's a big unit like a chariot or a trebuchet, and we want to build a derived map that stores other properties, say, whether a tile has a unit in it, or a filter by unit health. Maps like these are meant for various fast checks: can you move to this point, which units are worth attacking. Data is usually represented with some integer value, say, a unit's index within a tile. As a sample task, let's say we only want to check units whose health is above a certain value. That condition isn't picked at random, either; some units implement strategies like "kill the weakest" or "attack as a pack." In strategy games, walking through every nearby unit one by one (especially if every unit in a group is doing this) can turn into a seriously expensive operation. The algorithm for filling a bitmap looks something like this, say we want a picture of which units have health above a certain value:

```

array length: N
unitExist[x][y] = !(unitHealth[x][y] > limit)
for i = 0...N-1
    byte = pack (!(unitHealth[i] > limit),
                !(unitHealth[i+1] > limit)
                ...,
                !(unitHealth[i+7] > limit))
    output[i/8] = byte
    i+=8

```

Using bit packing also saves memory:

```

256 X 256 * 1 (byte/health) = 65536 (bytes) = 64kb
64kb/8 = 8192 (bytes) = 8kb

```

Working with bit sets like this requires different algorithms, and sometimes that extra work can meaningfully slow down the whole process. This problem resembles compression algorithms, though packing is usually a much faster process. These days, the tension between memory footprint and compute power isn't nearly as sharp as it was, say, twenty years ago, so it's up to the developer to decide what to trade for what. To have something to compare against, let's take a few well-established approaches, each with its own pros and cons: `std::bitset`; `std::vector`; one "manual" hand-rolled variant; community implementations

No Packing

```
struct NoPacking {
    vector<byte> flags;
    NoPacking(size_t size) : flags(size, false) {}
    void set(size_t index, bool value) {
        flags[index] = value;
    }
    bool get(size_t index) const {
        return flags[index];
    }
    size_t memory_usage() const {
        return flags.size() * sizeof(byte);    }
    void flip(size_t index) {
        flags[index] = !flags[index];
    }
    void bitwise_or(const NoPacking& other) {
        for (size_t i = 0; i < flags.size(); ++i) {
            flags[i] = flags[i] || other.flags[i];
        }
    }
};
```

This approach is just plain storage of boolean values with no optimization at all. Every boolean value takes up a full byte in memory, even though only a single bit of information is actually used. The implementation uses a standard *vector*, but treats it like an ordinary container, without leaning on anything specialized about its nature. Element access happens through simple indexing operations with no extra computation. It gives good theoretical performance, and maximum simplicity and readability. Read and write operations run at full speed, since there's no need for any bitwise operations or extra computation. Debugging code like this doesn't look hard, since every value can be traced easily in memory. There are no thread-safety issues of the kind that can crop up when packing several bits into one byte. The main downside is extremely inefficient memory use, since 87.5% of the space goes to waste (7 out of every 8 bits in each byte sit unused). With large data volumes, this leads to poor cache-line utilization and a significant slowdown from frequent main-memory access. The data-fill pattern for a container like this looks like this:

```

uhealths.reset(new int[N]);
exist.reset(new bool[N]);
length = N;

std::mt19937 gen(0);
std::uniform_int_distribution<> dist(0, 255);

for (size_t i = 0; i < testLength; ++i) {
    uhealths[i] = dist(gen);
    exist[i] = (uhealths[i] > limit);
}

```

std::bitset

This is a template class whose size is fixed at compile time. Internally, it uses an array of integers (usually unsigned long or unsigned long long) to store bits, packing them as tightly as possible. The compiler generates code for bit operations using built-in CPU instructions. Indexing works by computing the word number and the bit's position within that word, via division and modulo. It gives fairly solid performance among packing methods, thanks to compiler optimizations and a well-known packing pattern. Memory is used efficiently, since every bit of data occupies exactly one bit in memory. The main downside is that the size has to be known at compile time, which rules out building dynamic data structures and limits its use wherever sizes are only known at runtime. The data-fill pattern for a container like this looks like this:

```

for (size_t i = 0; i < length; ++i) {
    exist_bs.set(i, !(uhealths[i] > limit));
}

```

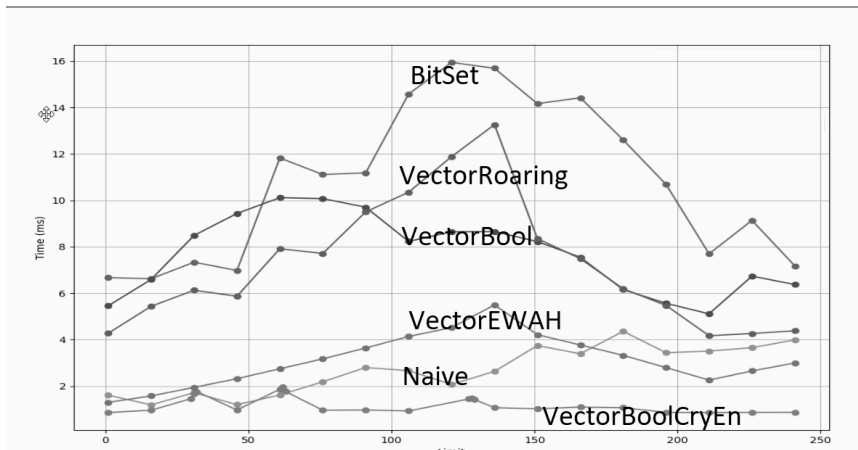
custom_bitset

In a hand-rolled implementation, the packed bits usually live in an array of integer values (*std::vector*). Each word holds N bits of data, and accessing a single bit means computing an index and a bit position within that segment. Setting and clearing bits happens through bitwise OR and AND with the appropriate masks. All of this gives you full control over the implementation and lets you optimize for your exact requirements. Using larger data types saves memory, and not depending on complex standard-library classes makes porting the code across platforms easier. But performance can suffer from all the work done with machine words. Having to implement every

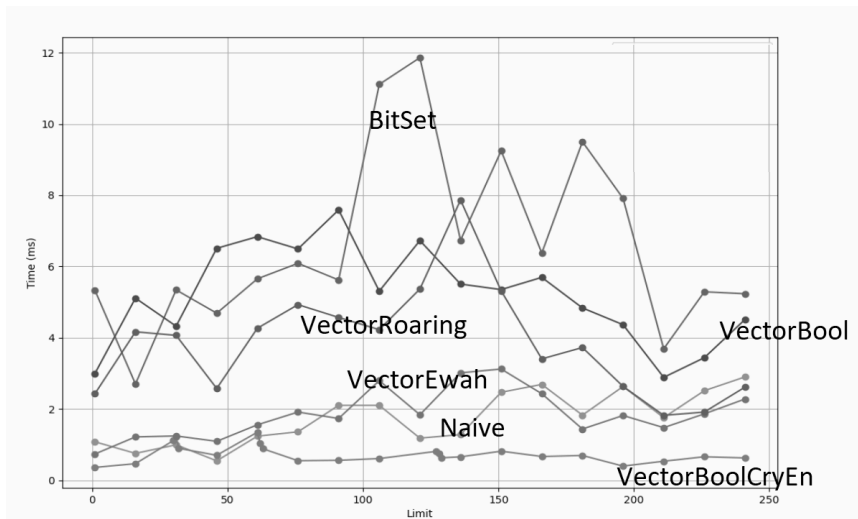
operation yourself increases both the odds of bugs and development time. The lack of automatic compiler optimizations that apply to standard containers can also hurt performance, and the code ends up less portable and more tied to the target CPU architecture. When hand-rolling bit arrays, the choice of base type size (`uint32_t`, `uint64_t`, or `uint128_t`) has a big effect on performance; `uint64_t` often turns out to be the sweet spot on almost every architecture, though on some ARM processors `uint32_t` runs faster because of quirks in the instruction pipeline. Interestingly, `uint128_t`, despite the theoretical memory savings, often ends up slower, because the processor may not have native support for 128-bit operations and has to emulate them across several instructions. The fill pattern for this container and others like it looks like this:

```
for (size_t i = 0; i < arrayLength; ++i)
{
    exist_vb[i] = !(uhealths[i] > health);
}
```

For the benchmark, a few more bit-manipulation library implementations were thrown in (you can find all of them on GitHub): `CRoaring`: implements the compressed Roaring format in C, with a C++ wrapper. Works with GCC, clang, and Visual Studio. `EWAHBoolArray`: implements the compressed EWAH format in C++. A C version of this library is partially included in Git, the tool everyone knows. Similar to WAH and Concise, but faster. `cbitset`: implements an uncompressed bitset in C. Works with GCC and clang. `Concise`: this C++ library implements the compressed WAH and CONCISE formats. Works with GCC and clang. `BitMagic`: this C++ library uses its own compression format. Somewhat similar to Roaring, but can use more memory. Works with GCC, clang, and Visual Studio. Benchmarking these on Intel/AMD processors produced two sets of graphs with some interesting patterns (figures on this page and the next).



Intel: for larger values around 127, you can see a drop in branch-misprediction counts, and a corresponding speedup, since the CPU can see from its BPU history that earlier branches went the other way. Nearly every algorithm shows its sharpest peak right around the value 128, which suggests something about the likely depth of the branch-history buffer. You can also see spikes around the 32, 64, and 128 marks across every algorithm tested. My best guess is that the problem comes down to branch mispredictions and frequent misses in the branch-prediction unit. When the threshold value is relatively stable, the odds that the input values will flip the branch are fairly low. But for thresholds that are multiples of, or close to, 32, the picture changes, and that depends on how deep the branch predictor's history buffer is and what conditions "pollute" it with data.



AMD: on this chart, the standout is a solution from the CryEngine codebase (VectorBoolCryEn), which delivers the best results on both processors, the bottom chart. The idea behind it is very simple: minimize the number of conditional branches. Instead, it uses the setg instruction (set byte if greater), which isn't a real branch at all. The algorithm is less prone to BPU (branch predictor) misses, because setg doesn't change the flow of control the way conditional branches do.

```
void bm_branch(int* uhealths, uint8_t* bm, int len, int
limit) {
    uint8_t mask[8] = { 0 };
    const auto lenDivBy8 = (len / 8) * 8;
    auto h = uhealths;
    auto bb = bm;
    for (int j = 0; j < lenDivBy8; j += 8) {
        mask[0] = h[0] > limit ? 0x01 : 0;
        mask[1] = h[1] > limit ? 0x02 : 0;
        mask[2] = h[2] > limit ? 0x04 : 0;
        mask[3] = h[3] > limit ? 0x08 : 0;
        mask[4] = h[4] > limit ? 0x10 : 0;
        mask[5] = h[5] > limit ? 0x20 : 0;
        mask[6] = h[6] > limit ? 0x40 : 0;
        mask[7] = h[7] > limit ? 0x80 : 0;
        *bb++ = mask[0] | mask[1] | mask[2] | mask[3]
                | mask[4] | mask[5] | mask[6] |
mask[7];
        h += 8;
    }
}
```

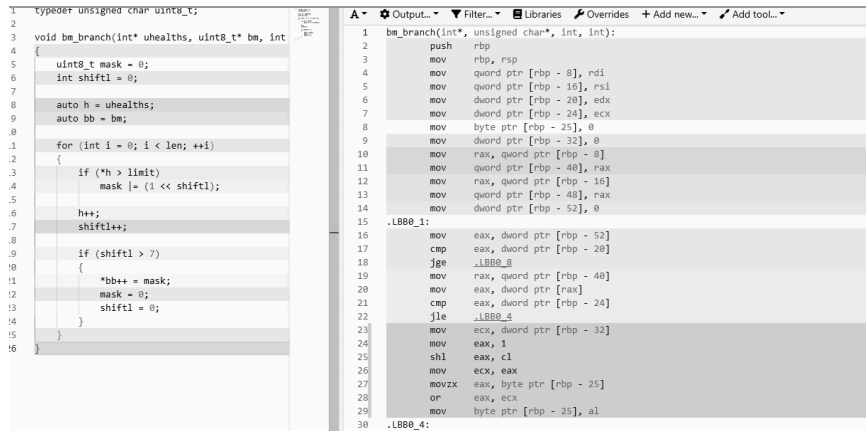
The naive version of the code (and I'd guess the other algorithms too) has branches inside the loop, which hurts performance (fig. 3.8). And when the processor hits an unpredictable branch, especially on fairly random data, it all adds up to a multiple-fold drop in performance:

```

void bm_branch(int* uhealths, uint8_t* bm, int len, int
limit) {
    // Current bit mask accumulating comparison results
    // Each bit represents the result of one comparison
    uint8_t mask = 0;
    int shiftl = 0; // Bit-position counter within the
current byte (0-7)
    // Used to decide which bit in mask needs to be set
    // Walk the uhealths array, which has len elements
    for (int i = 0; i < len; ++i) {
        // If the health value is above the limit, set the
bit to 1
        // Use bitwise OR to set the bit at position shiftl
        if (*uhealths > limit) { mask |= (1 << shiftl); }
        uhealths++; shiftl++;
        if (shiftl > 7) { // Check whether 8 bits of the
current byte are filled
            *bm++ = mask; // Write the filled bit mask to
bm
            mask = 0; // Reset the mask for the next byte
            shiftl = 0;
        }
    }
}

```

The same idea, but with branches, and the compiler couldn't optimize this code even with a well-known usage pattern. If you look at the original code, you'll notice an unrolled loop; one colleague suggested in review just widening the mask's range and handling more bits at once, which gives you a x16 version that runs faster, though it doesn't deliver any extraordinary speedup. Going a step further and implementing it with SSE instructions lets you process the integer array (*uhealths*), checking for each value whether it exceeds the threshold (set via the *limit* variable), and building an output array (*bm*) that stores the bit masks for each batch of values.



Manually vectorizing the array processing lets you handle 16 values per instruction (on consoles, that can stretch to 32 per operation with AVX2), speeding things up considerably compared to traditional loops. Cutting down the number of memory operations and getting rid of branches entirely gives the algorithm an extra speed boost:

```

// Compute the length rounded down to a multiple of 16 (we
// process 16 bytes at a time)
// The remainder (testLen % 16) gets handled separately by
// scalar code
const size_t lenDiv16y16 = (testLen / 16) * 16;
// Build an SSE constant with the value 127 in all 16
// bytes
// Used to convert from the unsigned range into the signed
// range
const __m128i sse127 = _mm_set1_epi8(127);
const int8_t ConvertedThreshold = limit - 128;
const __m128i sseThresholds =
    _mm_set1_epi8(ConvertedThreshold);
// Pointers for reading the input data and writing the
// result
auto pInputData = pValues;
auto pOutputByte = pOutput;

```

```

// Process the data in blocks of 16 elements (128 bits)
for (size_t j = 0; j < lenDiv16y16; j += 16) {
    // Load 16 bytes from an unaligned memory address into
    an SSE register
    // _mm_loadu_si128 is an unaligned load, no 16-byte
    alignment required
    const auto in16Values = _mm_loadu_si128(pInputData);
    // Run a parallel comparison of all 16 bytes against
    the threshold
    // Result is 0xFF if greater, 0x00 if less than or
    equal
    // _mm_cmpgt_epi8 = compare greater than (signed 8-bit
    integers)
    const auto cmpRes = _mm_cmpgt_epi8(in16Values,
    sseThresholds);
    // Extract the top bit of each of the 16 bytes
    // Each result bit represents one element
    // This is the bitmask equivalent, handling 16 elements
    at once
    const auto packed = _mm_movemask_epi8(cmpRes);
    *((uint16_t *)pOutputByte) = static_cast<uint16_t>
    (packed);
    // Advance to the next 2 bytes of the output buffer (16
    bits = 2 bytes)
    pOutputByte += 2;
    // Advance to the next 16 elements of the input array
    pInputData += 16;
}

```

Conclusions

A comprehensive approach to optimizing bitmap processing can speed things up significantly, sometimes several times over. That comes from eliminating redundant branches, using the processor's vector instructions, and carefully tuning algorithms for a specific task. Conditional branches remain one of the most expensive things in a program, even on today's high-performance processors. That comes down to the instruction pipeline's architecture: whenever the processor hits a conditional branch, it has to predict which way execution will go. A misprediction flushes the pipeline, which causes significant delays, sometimes dozens of processor cycles. That said, it's worth understanding that branch optimization only pays off in genuinely critical code paths. In most ordinary programs, the overhead of doing this kind of optimization outweighs the benefit. Fighting branches is

only worth it in heavily loaded logic. Pay attention to branches specifically in so-called "hot" functions, the code paths that run most often and eat up a meaningful share of the program's runtime. That's exactly where branch optimization delivers the biggest payoff. Modern profilers help you find spots like these and show where the processor is burning time on branch mispredictions. Don't automatically swap out your own implementations for standard-library solutions in hopes of better performance. The `std::bitset` algorithm in C++ performs worse than a hand-written bit-operation implementation tailored to a specific task. Standard libraries are built for general-purpose use and need to behave correctly across a huge range of scenarios. That inevitably means trading off some performance. A custom implementation, by contrast, can be tuned precisely to specific requirements: data size, access patterns, operation frequency, and the target processor's architecture. As a result, a "reinvented wheel" often ends up faster than the standard solution. That said, with C++20, we're getting more standard bit-manipulation tools of our own:

```
std::uint8_t num= 0b00110010
std::cout << "std::has_single_bit" <<
std::has_single_bit(num);
std::cout << "std::bit_ceil" << std::bitset<8>
(std::bit_ceil(num))
std::cout << "std::bit_floor" << std::bitset<8>
(std::bit_floor(num))
std::cout << "std::bit_width(5u)" << std::bit_width(5u)
std::cout << "std::rotr(0b00110010, 2)" << std::bitset<8>
(std::rotr(num, 2))
std::cout << "std::rotr(0b00110010, 2)" << std::bitset<8>
(std::rotr(num, 2))
std::cout << "std::countl_zero" << std::countl_zero(num)
std::cout << "std::countl_one" << std::countl_one(num)
std::cout << "std::countr_zero" << std::countr_zero(num)

std::has_single_bit(0b00110010): false
std::bit_ceil(0b00110010): 01000000
std::bit_floor(0b00110010): 00100000
std::bit_width(5u): 3
std::rotr(0b00110010, 2): 11001000
std::rotr(0b00110010, 2): 10001100
std::countl_zero(0b00110010): 2
std::countl_one(0b00110010): 0
std::countr_zero(0b00110010): 1
```

End of the Free Sample

Thank you for reading the first part of *Game++*. This sample covers the front matter, Part 0x1 and the Optimizations chapter — about a fifth of the book.

The full edition continues with:

Part 0x2 — Low-Level Optimizations

- 0x21 — String Optimizations
- 0x28 — Array Optimizations
- 0x2A — Allocator Optimizations
- 0x2D — Container Optimizations
- 0x2E — Heap of Fun

Part 0x3 — The Standard Library

- 0x30 — STL and Algorithms

Part 0x4 — Concurrency

- 0x40 — Multithreaded Programming

Part 0x5 — Design Patterns

- 0x50 — Design Patterns in Actions

Get the full book

<https://dalerank.github.io/#books>