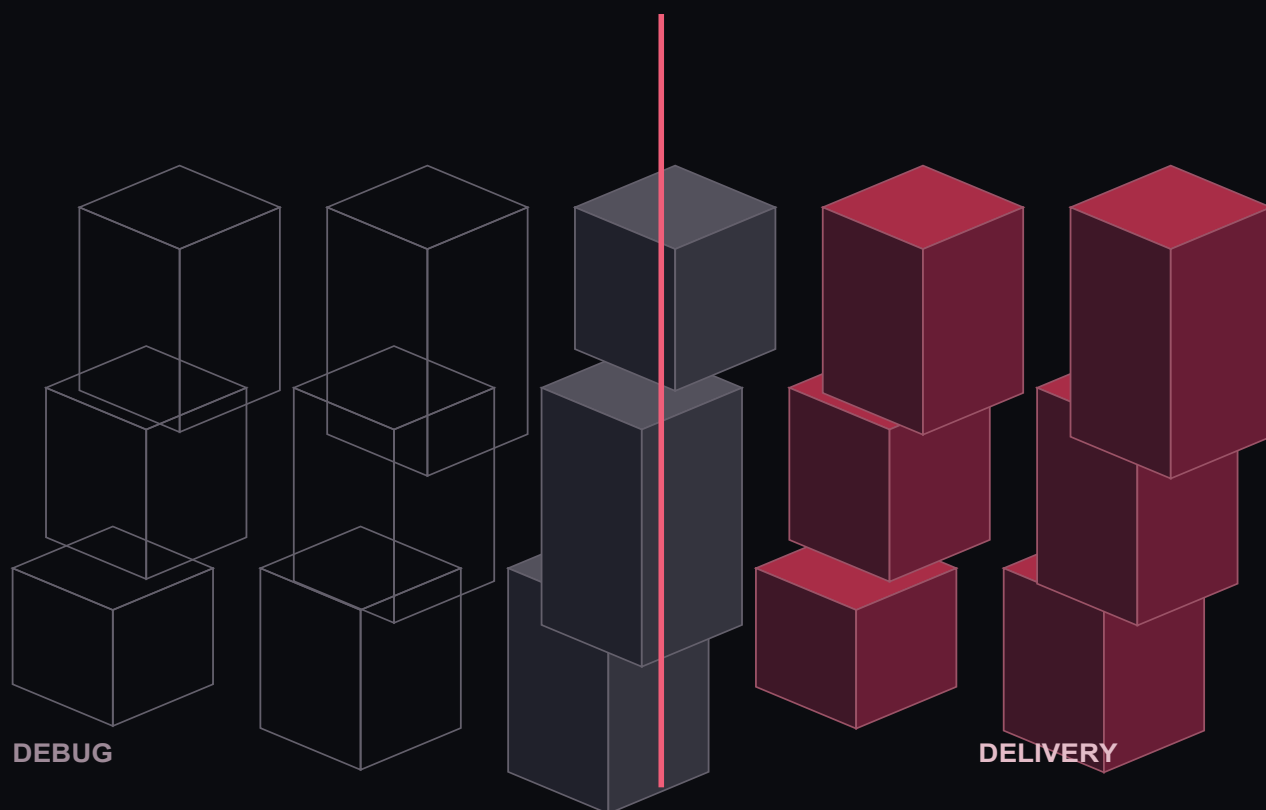


THE INDIE GAME OPTIMIZATION HANDBOOK

Build Bigger Worlds Without Destroying Performance



UNITY & PERFORMANCE

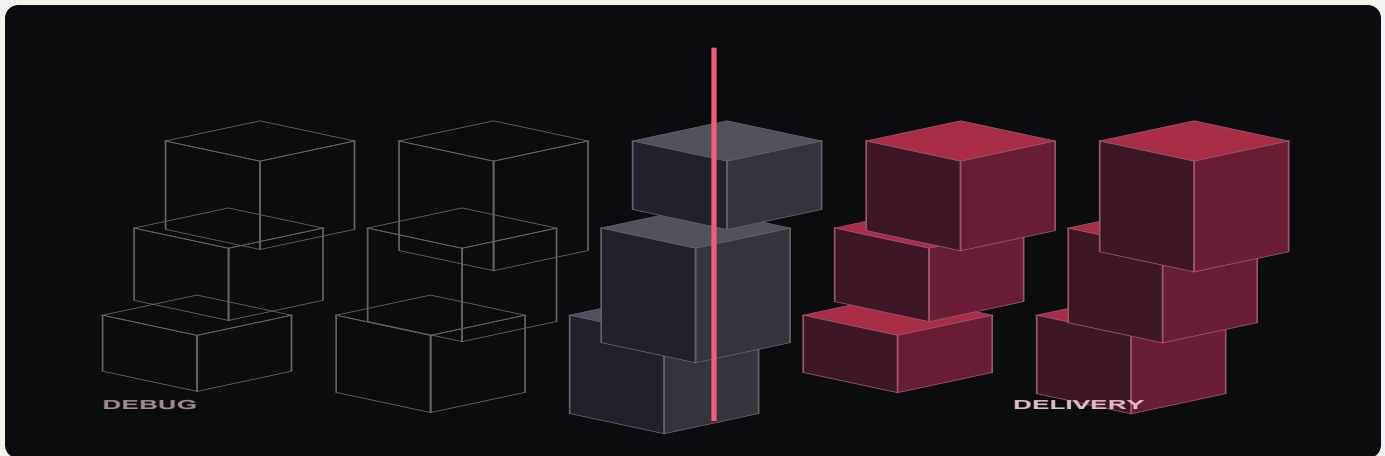
BUILD WHAT YOU IMAGINE.

FIELD GUIDE

A PRACTICAL FIELD GUIDE

The Indie Game Optimization Handbook

Build Bigger Worlds Without Destroying Performance



Published by Vertex Arc

First edition / October 2026

Original text and procedural illustrations created for Vertex Arc. Copyright 2026 Vertex Arc. Free to download and read. All examples describe hypothetical projects; they do not claim client work or shipped results.

Use this guide as a set of practical experiments. Test advice in your own project and consult documentation for the engine version you ship. Game-design observations are design hypotheses, not clinical claims or guarantees. No affiliation with or endorsement by Unity Technologies is implied.

vertexarc.site/books

Inside this guide

Read in sequence, or choose the lesson that answers the next question in your project. Page numbers refer to the PDF page counter.

Introduction: protect the experience with evidence	4
01 Profile before optimizing	5
02 Distinguish CPU and GPU pressure	6
03 Draw calls, batching, and material discipline	7
04 LOD and visibility should preserve the scene	8
05 Textures and lighting need explicit budgets	9
06 Physics and AI should work at the right frequency	10
07 Garbage collection and pooling have tradeoffs	11
08 Memory and loading are a single journey	12
09 Mobile, WebGL, and PC need different evidence	13
10 Verify the fix and prevent the regression	14
Troubleshooting: choose the next experiment	15
Field checklist: performance review	16
Conclusion: make performance part of production	17
Technical references	18

About Vertex Arc / 19

Introduction: protect the experience with evidence

Optimization is the process of changing a measured cost while preserving the behavior and presentation that matter. It is not a collection of switches that always improve performance. A technique can help one project, do nothing in another, or move work into a different bottleneck. Start with a specific build, device, scenario, and target.

This book follows an original hypothetical Unity project called Relay Valley: a compact exploration game with an outdoor route, a busy machinery yard, and an interior control room. The example gives us different visibility, simulation, and loading conditions. Any numbers used to explain budgets are illustrative, not measured results from a shipped Vertex Arc game.

Keep three artifacts: a repeatable test route, a baseline capture, and a change log. Record build version, resolution, quality settings, device, and test conditions. Compare changes under similar conditions. A faster run on a cooler phone or a different scene state is not a fair comparison.

The workflow is simple to state: reproduce, measure, form a hypothesis, change one cause, and verify. The difficult part is resisting a familiar optimization before you know whether it addresses the problem. Use this book as a diagnostic companion. Read the chapter that matches the evidence, then return to the same test and check the result.

THE OPTIMIZATION LOOP



Use the result to shape the next attempt.

01 / Profile before optimizing

THE IDEA

Describe the symptom precisely. Does the game run slowly throughout an encounter, hitch when a new effect appears, or consume more memory after every transition? These are different problems. Capture the event on the intended platform where possible. The Unity Editor performs additional work, and development instrumentation can also affect results.

Use frame time alongside frame rate. A 60 FPS target corresponds to roughly 16.7 milliseconds per frame, but an average can conceal occasional long frames. Inspect the timing around visible hitches and distinguish steady costs from first-use costs. Preserve the trace before making changes so later comparisons have a reference.

EXAMPLE

Relay Valley pauses briefly when the player first enters the machinery yard, then runs smoothly. Reducing every prop's polygon count might lower sustained rendering cost without fixing the entry hitch. The investigation instead captures the transition and separates asset preparation from ongoing simulation. The fix follows the measured event, not the most visually complicated part of the scene.

KEY TAKEAWAY

Name the scenario and capture the cost before choosing a technique. A repeatable baseline is part of the optimization work.

TRY THIS

Create a one-minute route with a quiet area, a busy encounter, and a transition. Capture it on a target device. Mark the most expensive sustained interval and the longest visible hitch. Write a separate hypothesis for each. Do not change both at once.

DEVELOPER NOTE

Deep profiling can add substantial overhead. Use detailed instrumentation to investigate a narrowed question, then validate the final result with representative profiling and a release-like build. Record the capture mode with the result.

02 / Distinguish CPU and GPU pressure

THE IDEA

The CPU prepares and simulates work; the GPU executes rendering work. Their activity overlaps, and one can wait for the other. A long wait marker is not automatically the original cause. Use profiler context and controlled experiments to understand the limiting path before interpreting a single number.

Changing resolution can help test whether pixel-related rendering work matters, but it is a clue rather than a complete diagnosis. A CPU-heavy scene may show little improvement when resolution drops. A GPU-heavy effect may improve substantially. Verify the result with the relevant profiler tools and keep the scene state comparable.

EXAMPLE

The yard contains many active machines and transparent steam effects. Disabling machine logic in a test build helps little, while temporarily reducing steam coverage improves the measured slow interval. This points toward rendering work worth investigating. It does not prove that every transparent effect is bad or that CPU work needs no future attention.

KEY TAKEAWAY

Use controlled changes to narrow the bottleneck. Do not confuse the place a system waits with the work that made it wait.

TRY THIS

Capture the same scene at two resolutions, then with one suspected effect temporarily disabled. Compare CPU and GPU timings, not only the FPS counter. Restore the original settings after the experiment and write what the evidence supports. If results are inconsistent, improve the test before implementing a permanent change.

DEVELOPER NOTE

VSync, frame caps, thermal conditions, and background activity can affect interpretation. A capped game may have useful headroom that an FPS counter does not reveal. Document these conditions rather than silently changing them between tests.

03 / Draw calls, batching, and material discipline

THE IDEA

Rendering many objects can create CPU submission work and state changes. Batching and instancing approaches can reduce some costs under suitable conditions, but their behavior depends on the render pipeline, shaders, materials, geometry, and platform. A lower draw-call count is not a complete performance result by itself.

Use the Frame Debugger and profiler to see how the actual scene is rendered. Look for avoidable material variants and objects that break an otherwise useful rendering pattern. Preserve clear ownership of materials so a small visual adjustment does not silently create hundreds of unique instances.

EXAMPLE

Relay Valley uses the same pipe module throughout the yard, but each instance has a separately duplicated material. The team tests a shared material approach and an appropriate instancing path for its pipeline. It measures CPU and GPU costs before choosing the production setup. Combining every pipe into one enormous mesh is avoided because it would reduce visibility granularity and complicate editing.

KEY TAKEAWAY

Optimize the measured rendering work, not a counter in isolation. Fewer submissions can still come with memory or visibility tradeoffs.

TRY THIS

Inspect a representative frame and identify repeated geometry with similar appearance. Check why it is rendered as it is. Test one consolidation or instancing change, then compare timing, memory, and culling behavior from several viewpoints. Keep the asset workflow understandable for the people authoring the scene.

DEVELOPER NOTE

The SRP Batcher reduces certain CPU setup costs for compatible shaders; it should not be described as a guarantee that objects become one draw call. Check the current documentation for your pipeline and Unity version before selecting a batching strategy.

04 / LOD and visibility should preserve the scene

THE IDEA

Level of detail systems reduce work when an object occupies less of the screen. The useful simplification depends on silhouette, material cost, animation, and viewing conditions. A distant object with a recognizable outline may tolerate substantial surface simplification, while a thin structure can change appearance noticeably when reduced.

Occlusion culling addresses objects hidden by other geometry. It is most useful when the scene provides meaningful occluders and the setup cost is justified. An open landscape and a dense interior create different opportunities. Test visibility changes from moving cameras and unexpected routes, not only the ideal viewpoint.

EXAMPLE

Relay Valley's control building hides much of the machinery yard from an interior camera. The outdoor ridge does not. The team tests occlusion for the building and LODs for distant machines rather than expecting one system to solve both cases. LOD transitions are tuned around actual camera distance and screen coverage.

KEY TAKEAWAY

Reduce invisible or insignificant work while preserving the shapes and transitions players notice.

TRY THIS

Choose one expensive repeated asset. Create a simpler distant version and test the transition during movement. Then inspect a representative interior for hidden geometry still contributing work. Compare costs with the relevant visibility approach enabled and disabled. Look for popping, missing objects, and new CPU overhead.

DEVELOPER NOTE

Do not merge geometry so aggressively that one visible corner forces a large hidden area to render. Asset organization, culling granularity, and authoring convenience need to be considered together.

05 / Textures and lighting need explicit budgets

THE IDEA

Texture resolution, format, mipmaps, and asset lifetime affect memory and bandwidth. Match resolution to the surface's importance and screen coverage. A source image's file size does not describe its runtime memory cost. Check imported platform data and test the result at gameplay distance.

Lighting choices also create authoring and runtime costs. Dynamic shadows, large light ranges, complex materials, and screen-space effects can become expensive in combination. Baked lighting has different constraints and memory requirements. Choose a setup that fits the game's changing objects and visual goals, then profile representative views.

EXAMPLE

Small warning labels in Relay Valley use excessive texture resolution while the player's main tool needs sharper detail. The team reallocates quality rather than reducing everything equally. It also shortens the range of background shadow-casting lights while preserving the light that communicates an important machine state.

KEY TAKEAWAY

Spend visual resources where they support identity and readability. Asset quality should follow use, not a universal maximum setting.

TRY THIS

Inspect the largest resident textures and identify how much screen space they actually occupy. Test lower resolutions or appropriate platform formats on a small set. For lighting, disable one suspected cost at a time in a capture build. Compare both the image and frame timing before making the change permanent.

DEVELOPER NOTE

Keep source assets at useful authoring quality and control runtime variants through a documented import pipeline. Destructively shrinking the only source file makes later revision harder and does not replace platform-specific configuration.

06 / Physics and AI should work at the right frequency

THE IDEA

Not every object needs expensive work every frame. Physics interactions, perception checks, path requests, and decision updates should follow gameplay requirements. Reducing frequency can save work, but it can also delay reactions. Define which behaviors need immediate response and which can tolerate scheduled updates.

Inspect collision layers, collider complexity, and the number of active interactions. Use simple shapes where they represent gameplay adequately. For AI, separate perception, planning, and movement so a full decision process is not repeated merely to continue an existing action. Handle unavailable paths and stale information deliberately.

EXAMPLE

Relay Valley's distant service drones do not need a new route calculation every frame. They follow an existing route and request a replacement when their destination or path validity changes. Nearby hazards retain timely collision behavior. The optimization reduces redundant decisions without making critical threats react late.

KEY TAKEAWAY

Reduce unnecessary frequency and scope before making systems more complicated. Preserve the response time that gameplay requires.

TRY THIS

Count perception queries and path requests during a busy scene. Identify work repeated with unchanged inputs. Schedule or cache one suitable operation, then test rapid player movement, blocked routes, and activation from off-screen. Check both cost and correctness under the worst transition.

DEVELOPER NOTE

Changing the physics timestep affects behavior and simulation cost. Do not treat it as a harmless quality slider. Test collisions, movement, and low-frame-rate catch-up behavior if the timestep changes.

07 / Garbage collection and pooling have tradeoffs

THE IDEA

Managed allocations can accumulate and contribute to garbage collection work. Inspect where allocations occur in the relevant scenario before replacing code. Frequent temporary collections, strings, or object creation may be worth addressing, but an allocation during a rare setup step is different from one repeated throughout combat.

Pooling keeps objects available for reuse instead of repeatedly creating and destroying them. It can reduce some churn while increasing retained memory and reset complexity. A pool is a lifecycle system: every reused object must return to a valid state, including events, timers, references, effects, and collision settings.

EXAMPLE

Relay Valley repeatedly spawns impact markers during a machinery test. A small measured pool helps that case. The team does not pool every object in the game. It defines a maximum retained count and a fallback for bursts, then tests that an old marker's timer cannot deactivate a newly reused instance.

KEY TAKEAWAY

Avoid repeated waste where it matters, but account for the memory and correctness cost of reuse.

TRY THIS

Capture allocations during a repeatable action sequence. Identify the largest recurring source and remove or reuse only that work. If you introduce a pool, test repeated acquire-release cycles, interrupted effects, scene changes, and demand above the pool's capacity. Compare peak memory as well as frame timing.

DEVELOPER NOTE

Do not promise that pooling eliminates garbage collection or guarantees smoother play. Other allocations and engine behavior remain. Incremental collection can distribute some work, but it does not make avoidable allocations free.

08 / Memory and loading are a single journey

THE IDEA

Memory must be understood over transitions and repeated sessions, not only in a steady scene. Old and new content may overlap during loading. Persistent managers can retain assets unintentionally. A system that looks stable on the first visit may grow after each return.

Define ownership and release points for loaded content. Asynchronous loading can improve responsiveness, but the game still needs a clear state while work is incomplete. Avoid exposing half-initialized gameplay or allowing duplicate requests to start overlapping loads. Progress indicators should reflect what can actually be measured.

EXAMPLE

Relay Valley loads the control room before unloading all outdoor content to avoid a blank transition. That overlap creates a peak. The team measures the peak and changes which assets must coexist, rather than judging only the final interior memory use. A return trip confirms that the first area's assets are not duplicated.

KEY TAKEAWAY

Measure the highest cost along the loading journey. A low steady-state number can hide an unsafe transition.

TRY THIS

Run a loop through three scenes several times and capture memory at the same checkpoints. Look for a stable pattern or continuing growth. Trace retained references and asset ownership before forcing cleanup. Test an interrupted or failed load and verify that retry returns to a usable state.

DEVELOPER NOTE

Avoid calling broad cleanup operations every frame or transition without evidence. They can introduce their own stalls. Fix lifetime and ownership first, then use deliberate cleanup at appropriate boundaries when the engine and asset system require it.

09 / Mobile, WebGL, and PC need different evidence

THE IDEA

Platform labels cover many devices. Mobile testing needs sustained sessions, thermal behavior, interruptions, and realistic memory limits. Browser testing includes download, decompression, startup, browser capabilities, and hosting configuration. PC testing includes a meaningful hardware range and settings that scale coherently.

Do not assume editor behavior proves any of these delivery paths. Build early for each required target and keep a representative test matrix. A platform-specific issue discovered after content production may require changes across assets, interfaces, and systems rather than a single export setting.

EXAMPLE

Relay Valley runs smoothly on a desktop but warms a phone during a long machinery encounter. Its web build also loads too much optional audio before the first interaction. These are separate investigations. A mobile quality tier reduces measured sustained rendering work, while the web delivery plan separates startup assets from later content.

KEY TAKEAWAY

Validate the complete player journey on the actual target. One powerful development device cannot stand in for every platform.

TRY THIS

Choose one minimum-spec device or browser environment per target. Test initial load, a busy interval, a transition, and a longer session. Record interruptions and recovery. On PC, compare quality presets for both performance and readable gameplay. On the web, inspect real transferred bytes and caching after an update.

DEVELOPER NOTE

Unity web capabilities vary by engine version and browser. Consult the current technical limitations for the version being shipped. Do not assume desktop threading, networking, storage, or plugin behavior transfers unchanged to a web build.

10 / Verify the fix and prevent the regression

THE IDEA

A change is not complete when the profiler looks better once. Repeat the original scenario under comparable conditions and test nearby behavior that may have changed. Record the before and after evidence, including any cost moved into memory, loading, or another frame. Keep changes that improve the intended experience, not merely an isolated metric.

Create a small milestone suite: a busy encounter, a camera route, a transition, and a sustained session. The suite should be practical enough to run regularly. Document its setup so another contributor can reproduce it without relying on the original developer's memory.

EXAMPLE

Relay Valley reduces a repeated AI query, but a drone now reacts late when activated near the player. The timing improvement is real, yet the change is not ready. The team keeps scheduled updates for distant agents and performs an immediate refresh on activation. The same performance test and the activation case are both rerun.

KEY TAKEAWAY

An optimization must pass performance and behavior checks. Record the tradeoff and make the test repeatable.

TRY THIS

Take one recent optimization and write a short acceptance note: original symptom, hypothesis, change, measured result, and regression cases. Ask a teammate to reproduce the result. If they cannot, improve the test instructions before treating the change as proven.

DEVELOPER NOTE

Common mistakes include optimizing from editor FPS alone, changing several variables together, ignoring memory, merging everything, pooling everything, and postponing target-device builds. Each replaces a specific investigation with a broad assumption.

Troubleshooting: choose the next experiment

SLOW THROUGHOUT A BUSY SCENE

Capture CPU and GPU timing in that exact scene state. Compare a controlled resolution change and one suspected system disabled. Check whether the limiting resource changes. Do not interpret a frame cap as proof that every subsystem has enough headroom.

A HITCH WHEN SOMETHING APPEARS FOR THE FIRST TIME

Compare the first use with later uses. Capture asset preparation, shader-related work where relevant, allocation, and initialization around the event. Test a planned preparation step, then verify that it does not merely move the hitch to a worse moment.

MEMORY GROWS AFTER EVERY LEVEL CHANGE

Repeat the same route and inspect retained content at equivalent checkpoints. Look for persistent references, duplicate loads, unbounded pools, and event subscriptions. A single forced cleanup can hide the pattern without correcting ownership.

PERFORMANCE FALLS DURING A LONG MOBILE SESSION

Keep scene state and settings consistent while recording device conditions. Investigate sustained CPU and GPU load and thermal behavior. Compare a coherent lower quality tier. Verify that input, pause, and resume remain dependable.

THE WEB BUILD STARTS SLOWLY

Inspect actual network transfers, compression headers, caching, and startup assets. Separate download time from decompression and initialization. Test a clean cache and an update from the previous build so mixed versions do not masquerade as a game bug.

Field checklist: performance review

- Define target hardware, quality settings, resolution, and frame-time goal.
- Record a repeatable baseline in a target build.
- Separate steady costs, first-use hitches, and transition peaks.
- Identify the limiting resource before choosing a technique.
- Inspect rendering submissions and material variants in a representative frame.
- Test LOD and occlusion from moving and unexpected viewpoints.
- Match texture and lighting budgets to visible importance.
- Reduce redundant physics and AI work without delaying essential reactions.
- Check allocation sources and complete reset behavior in pools.
- Repeat loading routes while watching memory ownership.
- Validate sustained mobile, deployed web, and scalable PC behavior as applicable.
- Preserve before-and-after evidence and rerun correctness checks.

Conclusion: make performance part of production

Performance work is most effective when the team can reproduce the problem and explain why a change addresses it. That clarity helps artists, designers, and programmers make compatible decisions. Budgets become useful coordination tools instead of arbitrary restrictions introduced near release.

Keep the smallest practical set of measurements alive throughout production. Test new content against representative scenarios, investigate regressions while the change is fresh, and preserve the visual and mechanical information players need. Bigger worlds are only useful when the experience remains dependable inside them.

Technical references

These official references support API and engine-specific details. The workflow examples and explanations in this book are original. Select the documentation version matching the project before applying a setting.

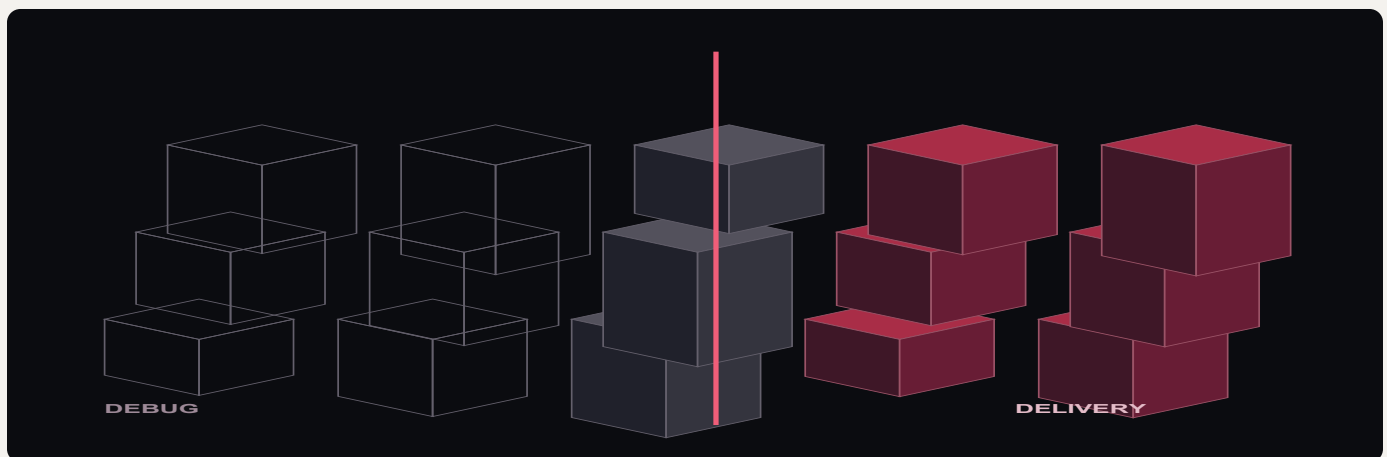
- Unity Profiler: <https://docs.unity3d.com/Manual/Profiler.html>
- Draw-call batching: <https://docs.unity3d.com/Manual/DrawCallBatching.html>
- Managed memory optimization:
<https://docs.unity3d.com/Manual/performance-optimizing-code-managed-memory.html>
- Web platform limitations: <https://docs.unity3d.com/Manual/webgl-technical-overview.html>

About Vertex Arc

BUILD WHAT YOU IMAGINE.

Vertex Arc is an independent game-development studio and creative engineering partner. Its work spans playable prototypes, game systems, and interactive experiences across PC, mobile, and web. This publishing series shares practical ways to ask better questions, make focused experiments, and improve the experience players actually receive.

The guides are designed to be used beside a project. Keep the exercises small, record what you observe, and adapt the next step to the evidence. Finishing one useful experiment is a better result than collecting advice that never reaches a playable build.



[Browse the complete free library](#)

[Explore the studio and its work](#)

[Start a project with Vertex Arc](#)