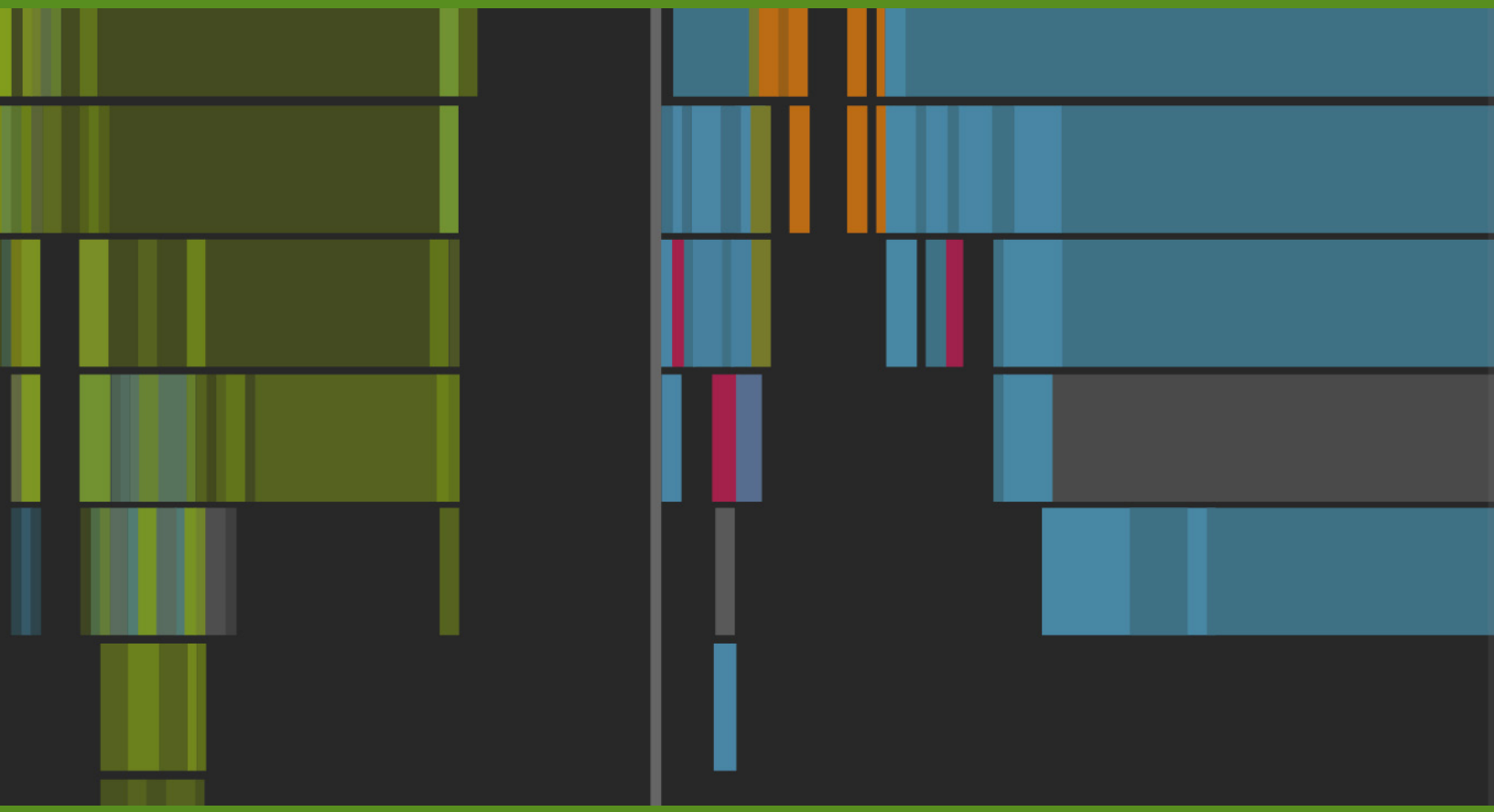
→ E-BOOK

Ultimate guide to profiling Unity games

(Unity 6 edition)



Unity®

Contents

Introduction

Smooth performance is essential to creating great gaming experiences that reach a broad range of devices and players. Unity provides a full set of profiling and memory management tools that Unity developers can use alongside the native profiling tools available for their target platforms.

This guide brings together actionable advice on how to profile an application in Unity, manage its memory, and optimize its power consumption from start to finish.

This second edition of the e-book has been updated to reflect the latest features in Unity 6, as well as improvements based on feedback from the community.

Our profiling guide was created as a collaboration between the following internal Unity experts and external game developer:

— Steven Cannavan, senior software development consultant

— Sean Duffy, software engineer and game developer

— Peter Hall, Senior Manager, Software Engineering

— Thomas Krogh-Jacobsen, senior manager, content marketing management

— Steve McGreal, software engineer

— Martin Tilo Schmitz, senior software engineer

— Peter Harris, Arm

Additional guides on performance optimization in Unity 6 include, Optimize your game performance for consoles and PC and Optimize your game performance for mobile, XR, and the web in Unity.

Profiling 101

Before diving into the details of how to profile a game in Unity, let's summarize some key concepts and profiling principles.

Lean, performant code and optimized memory usage lead to a better user experience across low- and high-end devices. This applies for everything, from being able to reach more users on the low-end devices by tackling heat and battery consumption, to your players' comfort levels, and ultimately, factors that drive higher adoption and retention. It can also be a requirement for passing distribution platform specifications.

A consistent, end-to-end profiling workflow is a "must have" for efficient game development; it starts with a simple three-point procedure:

— Profile before making major changes to establish a baseline.

— Profile during development to track and ensure changes don't break performance or budgets.

— Profile after to prove the changes had the desired effect.

Profilers are some of the most useful tools to have in your developer toolbelt for identifying memory and performance bottlenecks in your code.

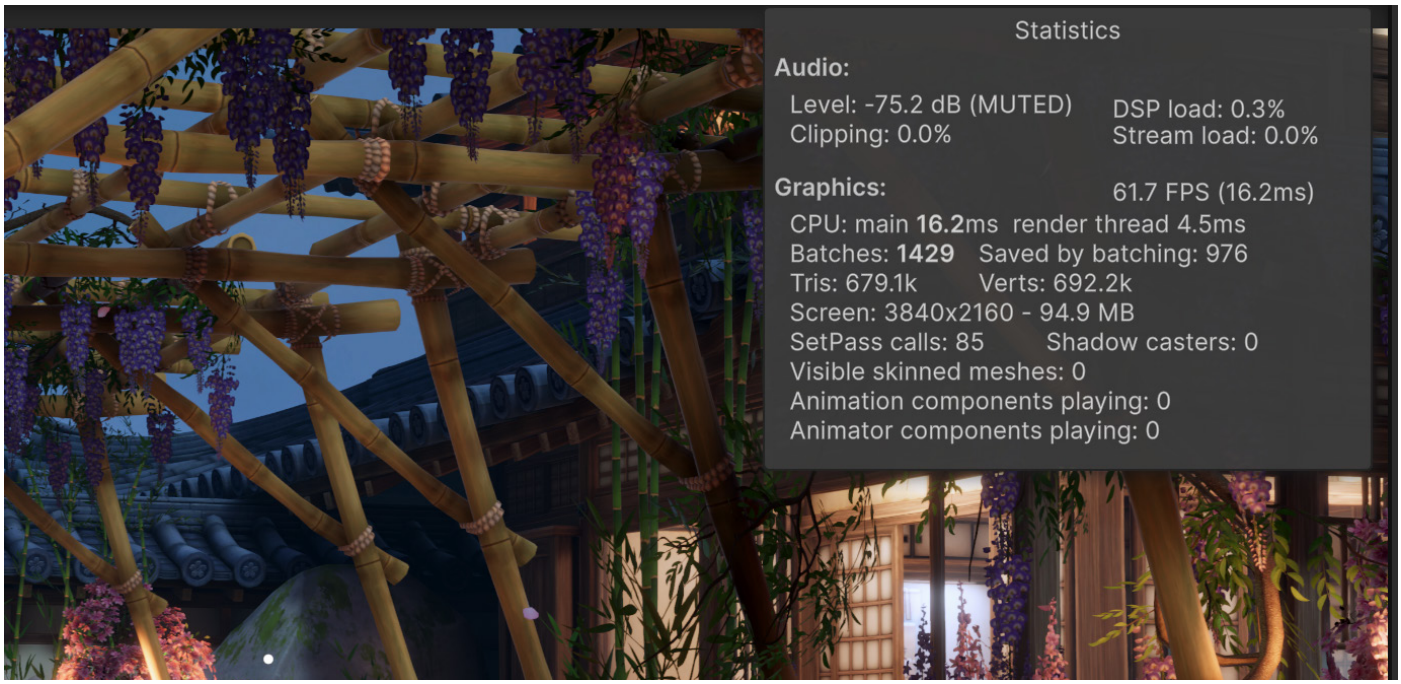
Think of profilers as detective tools that help you unravel the mysteries of why performance in your application is lagging or why code is allocating excess memory. They help you understand what is going on under the hood.

Unity ships with a variety of profiling tools for analyzing and optimizing your code, both in the Editor and on hardware. We'll dive into each of these in the e-book but it's also recommended to use native profiling tools for each target platform, be it mobile and other untethered devices, console, or PC.

Understanding frame budget

Gamers often measure performance using frame rate, or frames per second (fps), but as a developer it's generally recommended to use **frame time in milliseconds** instead. Consider the following simplified scenario:

During runtime, your game renders 59 frames in 0.75 seconds. However, the next frame takes 0.25 seconds to render. The average delivered frame rate of 60 fps sounds good, but in reality players will notice a stutter effect since the last frame takes a quarter of a second to render.



While Unity offers a variety of profiling tools for in-depth and precise analysis, you can also get a quick indication on the performance by simply looking at the Statistics panel in the Game view.

This is one of the reasons why it's important to aim for a specific time budget per frame. This provides you with a solid goal to work toward when profiling and optimizing your game, and ultimately, it creates a smoother and more consistent experience for your players.

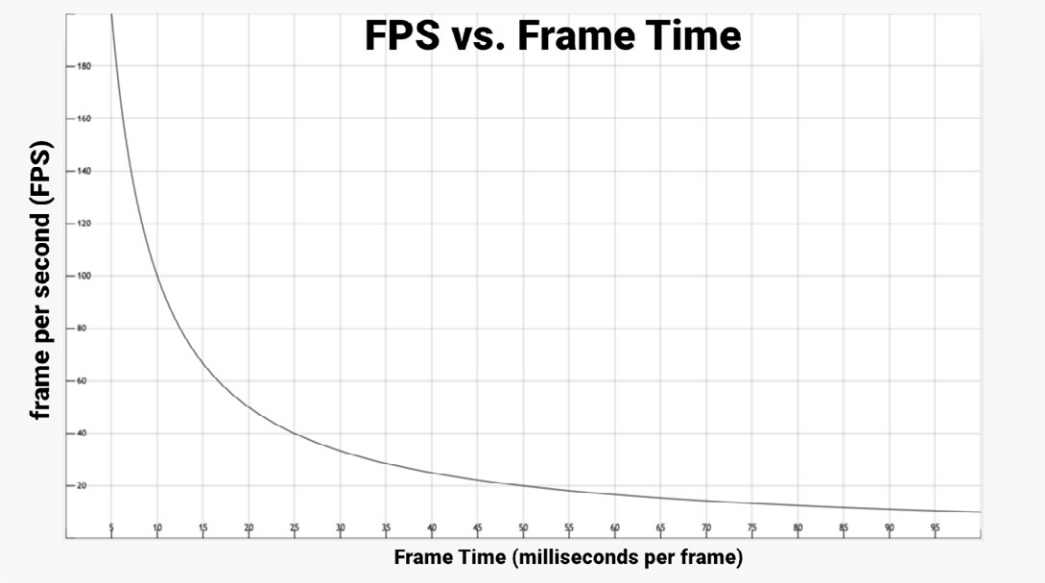
Each frame will have a time budget based on your target fps. An application targeting 30 fps should always take less than 33.33 ms per frame (1000 ms / 30 fps). Likewise, a target of 60 fps leaves 16.66 ms per frame.

You can exceed this budget during non-interactive sequences, when it's not disruptive for the immersive gaming experience, for example, when displaying UI menus or scene loading, but not during gameplay. Even a single frame that exceeds the target frame budget will cause hitches.

A consistently high frame rate in VR games is essential to avoid causing nausea or discomfort to players, and is often necessary for your game to get certification from the platform holder.

Frames per second: A deceptive metric

Why is it recommended that you measure frame time in milliseconds instead of frames per second? To understand why, look at this graph:



fps vs. frame time

Consider these numbers:

1000 ms/sec / 900 fps = 1.111 ms per frame
1000 ms/sec / 450 fps = 2.222 ms per frame

1000 ms/sec / 60 fps = 16.666 ms per frame
1000 ms/sec / 56.25 fps = 17.777 ms per frame

If your application is running at 900 fps, this translates into a frame time of 1.111 milliseconds per frame. At 450 fps, this is 2.222 milliseconds per frame. This represents a difference of only 1.111 milliseconds per frame, even though the frame rate appears to drop by one half.

If you look at the differences between 60 fps and 56.25 fps, that translates into 16.666 milliseconds per frame and 17.777 milliseconds per frame, respectively. This also represents 1.111 milliseconds extra per frame, but here, the drop in frame rate feels far less dramatic percentage-wise.

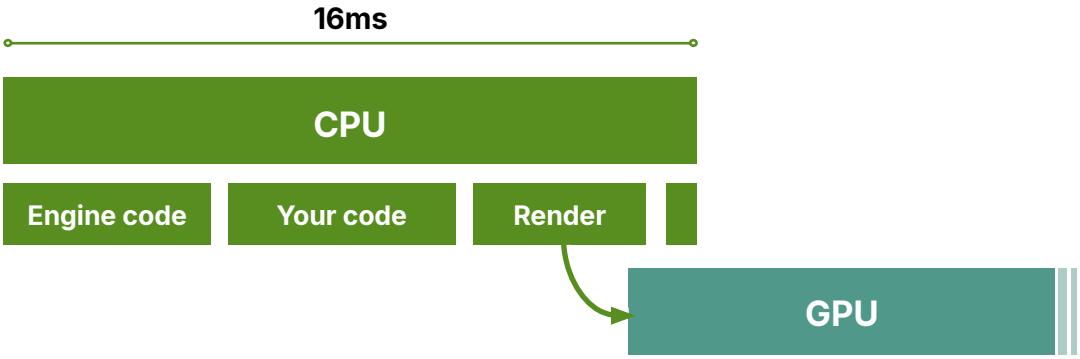
This is why developers use the average frame time to benchmark game speed rather than fps.

Don’t worry about fps unless you drop below your target frame rate. Focus on frame time to measure how fast your game is running, then stay within your frame budget.

Read Robert Dunlop’s article “FPS versus Frame Time,” for more information.

The anatomy of a frame

Let’s build on the above and explore how Unity constructs frames, and how the CPU (central processing unit) and GPU (graphics processing unit) collaborate during this process. While targeting 60 frames per second results in 16.66 milliseconds per frame, Unity actually maintains a pipeline where the CPU and GPU work on different frames simultaneously.



The CPU prepares rendering instructions that are handed off to the GPU.

On the CPU side, execution begins with Unity’s internal engine code (which is outside your control), followed by your custom game logic (your scripts). After this, the CPU prepares rendering instructions. These instructions are then handed off to the GPU; while the GPU begins rendering frame N, the CPU is already working on the following frame (denoted as N+1).

Unity uses a dual-threaded system to streamline this workflow:

— The main thread handles game logic, physics, animation, and input, while also queuing up rendering commands.

— The render thread converts these commands into GPU-friendly instructions.

Once the GPU receives the render instructions, it processes them through the graphics pipeline, performing tasks like vertex shading, fragment shading, post-processing, and finally, outputting the frame to the display.

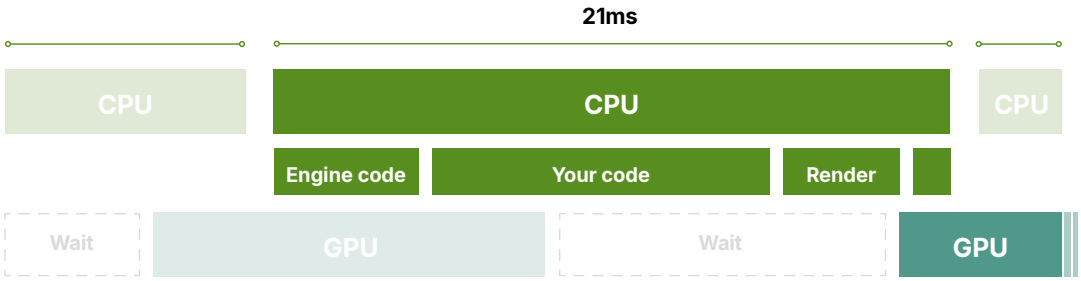
This parallelized approach allows the CPU to begin preparing the next frame while the GPU is still rendering the current one. However, the GPU must wait for the CPU to finish preparing rendering data before it can proceed, making synchronization between the two critical for performance.

To understand frame construction more deeply, refer to the event function execution order, which outlines the sequence of operations such as input handling, physics, rendering, and GUI updates that occur during each frame.

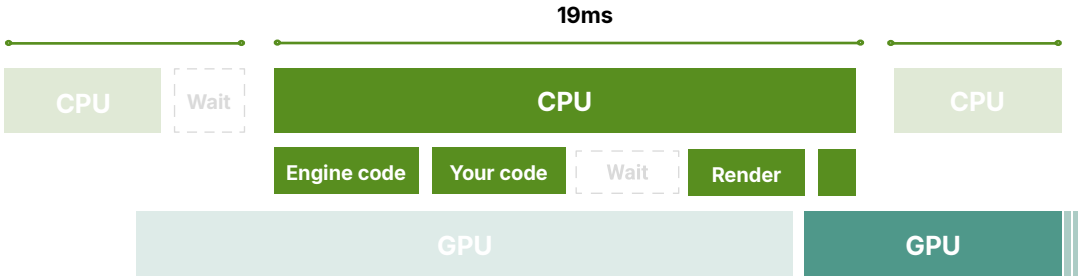
Understanding if you are GPU or CPU bound

CPU-bound vs. GPU-bound refers to which part of your system is limiting your game's performance and thus the key to understanding where to start your optimization journey.

CPU-bound means the CPU is the bottleneck. It's taking longer than the GPU to process tasks like scripts, physics, or draw call management. In this case, optimizing GPU settings won't improve frame rate. To remove the bottleneck you need to reduce CPU workload.

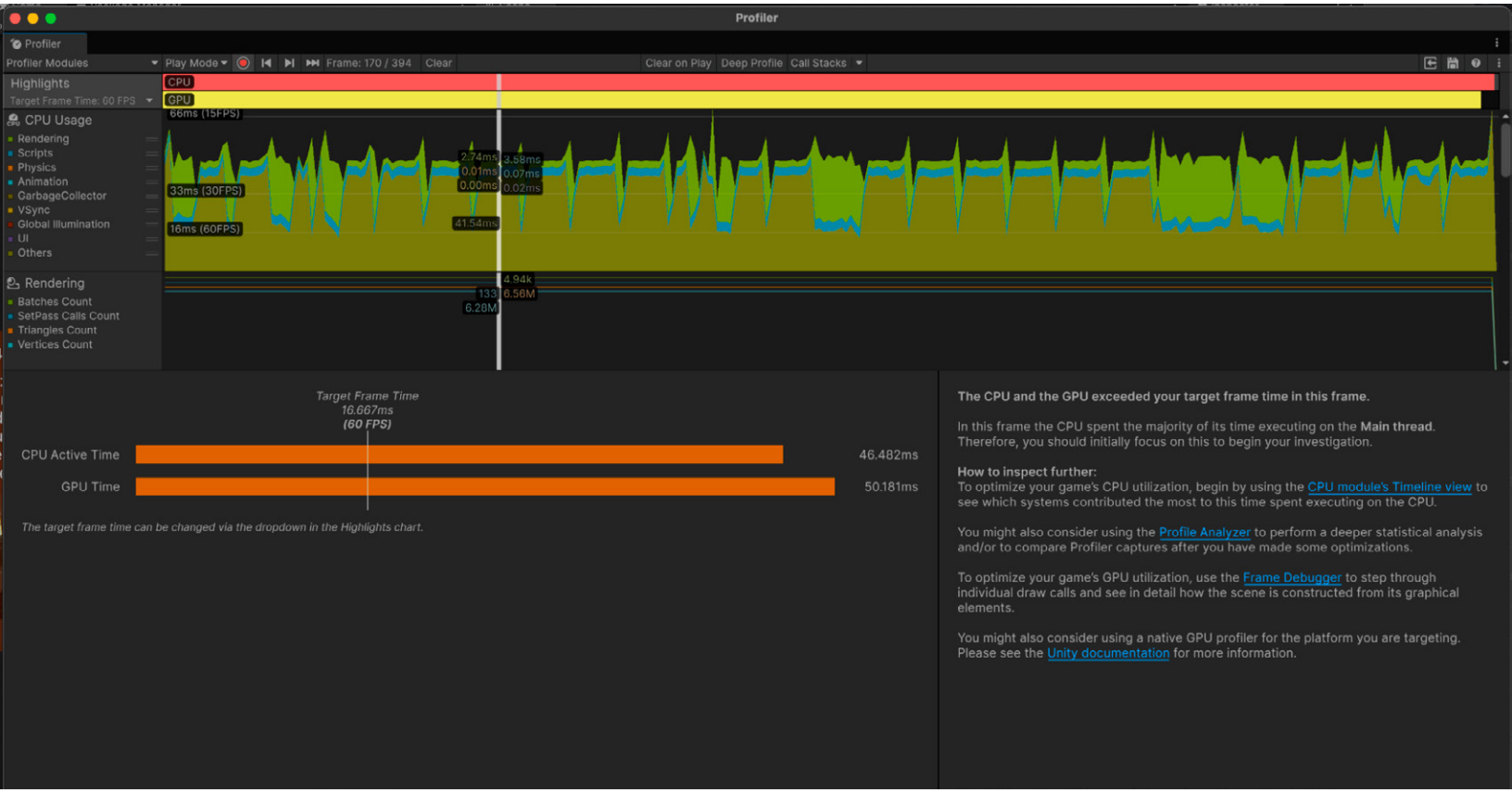


GPU-bound means the GPU is the bottleneck. It's spending more time rendering graphics than the CPU is spending on logic. Here, you'll want to simplify shaders, reduce lighting complexity, or lower resolution to improve performance, because optimizing your code on the CPU will not improve your framerate.



This wait is `Gfx.WaitForPresent` in the profiler

With the new Highlights module in Unity 6, it's easy to determine whether your application is CPU- or GPU-bound on all platforms.



The Highlights module makes it easy to understand how your game is performing vs the set target frame time. In this example, a lot of optimization work is needed on both the CPU and GPU to hit the target 60 fps.

What is VSync?

VSync synchronizes the application's frame rate with the monitor's refresh rate. This means that if you have a 60Hz monitor and your game runs within the frame budget of 16.66 ms, then it will be forced to run at 60 fps rather than allowed to run faster. Synchronizing your fps with your monitor's refresh rate lightens the burden on your GPU and stops visual artifacts such as screen tearing. In Unity, you can configure the **VSync Count** as a property in the Quality settings (**Edit > Project Settings > Quality**).

Understanding profiling in Unity

Unity's profiling tools are available in the Editor and via the Package Manager. These tools, along with the Unity Frame Debugger, are covered in more detail in the section titled "Unity profiling and debug tools", but here is a quick overview:

- The Unity Profiler measures the performance of the Unity Editor, and your application in Play mode or development mode while connected to a device.
- The Profiling Core package provides APIs that you can use to add contextual information to Unity Profiler captures.
- The Memory Profiler provides in-depth analysis of how much memory your game is using and what objects are using it.
- The Profile Analyzer enables you to compare two profiling datasets side by side to analyze how your changes affect your application's performance.
- The Project Auditor reports insights and issues about the scripts, assets, and code in your project, many of which relate to performance.

Unity also offers several debugging tools that complement its suite of profiling tools. The Rendering Debugger's Display Stats panel, for example, allows you to see a limited set of performance numbers and markers (CPU + GPU) on development builds without having the Editor connected.

Sample- vs instrumentation-based profiling

There are two common methods of profiling game performance:

- Sample-based profiling
- Instrumentation profiling

Sample-based profiling

Sample-based profiling works by taking periodic snapshots of what your code is doing at regular intervals (typically in milliseconds). By analyzing these snapshots, you can see which parts of your code are running most frequently and therefore consuming the most time. Generally, the overhead is low but the data is high-level as you are capturing aggregated snapshots. Increasing the sampling frequency can provide you with more data but without the precision that comes with instrumentation-based profilers.

Sampling profilers usually use platform infrastructure to provide minimum overhead and maximum sampling rate. Examples of such profilers are Windows Performance Analyzer in conjunction with Event Tracing for Windows, Instruments, and Android Studio.

Instrumentation-based profiler

Instrumentation-based profiling involves “instrumenting” the code by adding Profiler markers, which record detailed timing information about how long the code in each marker takes to execute. This profiler captures a stream of Begin and End event timestamps for each marker. It doesn’t lose any information, but it does rely on markers being placed in order for profiling data to be captured.

This allows you to explore the performance of your code, locate performance issues easily, and spot quick optimization wins, with the option of going even deeper by adding custom Profiler markers or using deep profiling. That also means the overhead is higher but that you are able to capture very precise information to identify specific issues compared to sample-based profiling.

Deep profiling automatically inserts Begin and End markers in every scripting method call, including C# Getter and Setter properties. This system gives full profiling detail on the scripting side, but it comes with an associated overhead that can inflate the reported timing data based on how many calls are within the captured profiling scopes.

Instrumentation vs sampling

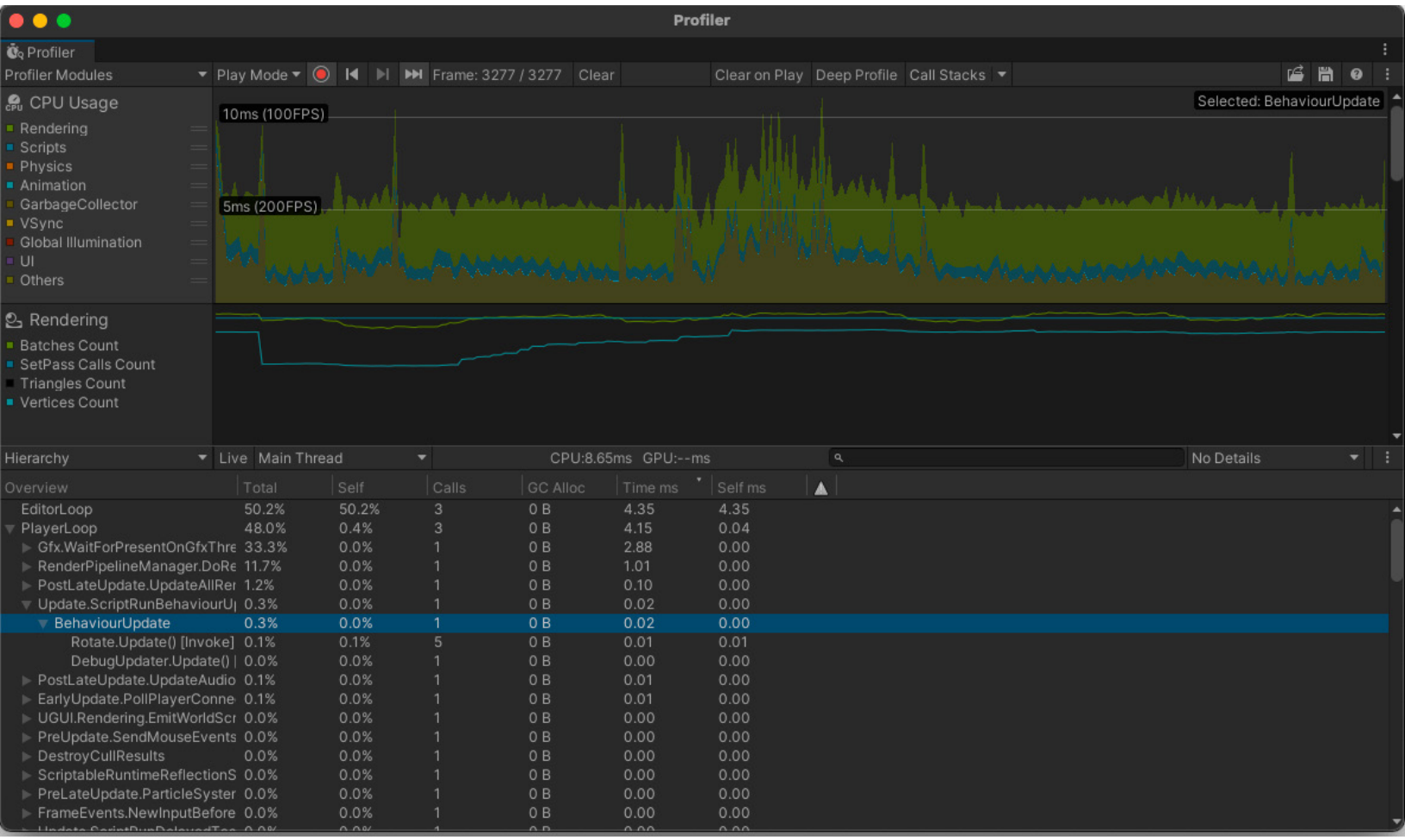
Generally, sample-based profiling analyzes the application’s high-level performance while instrumentation-based profiling pinpoints critical performance but with higher overhead.

The overhead introduced by sampling profilers is constant no matter the work that the CPU is doing while being profiled. You can change the sample rate to accommodate that but it’s generally lower. The overhead introduced by instrumentation profiling varies with the number of markers (i.e. adding a lot of markers will make the capture more expensive because the markers themselves take time). Just be aware of this when using instrumented profilers, because places in your project which call a lot of functions might look more expensive than they really are. This can show up especially when deep profiling and distorts the timings in your profiler capture.

Instrumentation-based profiling in Unity

The Unity Profiler combines both instrumentation-based and sample-based profiling, depending on the mode being used. A good balance of detail vs overhead is struck by markers being set in most of the Unity API surface. Important native functionality and scripting code base message calls are instrumented to capture the most important “broad strokes” without incurring too much overhead.

The scripting code base message calls mentioned above (instrumented explicitly by default) usually include the first call stack depth of invocations from Unity native code to your managed code. For example, common MonoBehaviour methods such as Start(), Update(), FixedUpdate(), and others are included.



Profiling an example script shows Update() method calls.

You can also see child samples of your managed scripting code that call back into Unity's API in the Profiler. However, one caveat is the Unity API code in question needs to have instrumentation Profiler markers itself. Most Unity APIs that carry performance overheads are instrumented. For example, using **Camera.main** will result in a **FindMainCamera** marker appearing in a profile capture. When examining a captured profiling dataset, it is useful to know what the different markers mean. Use this list of common Profiler markers to learn more.

Increase profiling detail with Profiler markers

By default, the Unity Profiler will profile code timings that are explicitly wrapped in Profiler markers. Manually inserting Profiler Markers into key functions in the code can be an efficient way to increase the detail level of profiling runs. Adding your own markers avoids incurring the full deep profiling overhead and the related problem of inaccurate times in your capture.

When deep profiling is enabled, Unity uses instrumentation-based profiling. This inserts additional instructions to collect precise, fine-grained data for every function call, making it possible to measure execution times and behaviors at the cost of higher runtime overhead.

Profiler modules

The Profiler captures per-frame performance metrics to help you identify bottlenecks. Drill down into details by using the modules included in the Profiler, such as **CPU Usage**, **GPU**, **Rendering**, **Memory**, **Physics**, and so on.

The main Profiler window shows the modules to the left and details panel at the bottom.

The Profiler window lists details captured with the currently selected Profiler module in a panel at the bottom of the view. The CPU Usage Profiler module, for instance, displays a **Timeline** or **Hierarchy** view of the work of the CPU, along with specific times.

The Timeline view available with the CPU Usage module; this view shows the Main and Render Thread marker detail

Use the Unity Profiler to assess your application's performance and dig into specific areas and issues. By default, the Profiler will connect to the Unity Editor Player instance.

Be aware that you will see a large difference in performance between profiling in the Editor and profiling a standalone build. Connecting the Profiler to a standalone build running directly on your target hardware is always preferable since this yields the most accurate results without Editor overhead.

Profiling workflow

This section looks at some useful goals when profiling, and common performance bottlenecks, such as being CPU-bound or GPU-bound. You'll learn how to identify these situations and investigate them in more detail. It also covers memory profiling, which is largely unrelated to runtime performance, but important to know about because it can prevent game crashes.

From high- to low-level profiling

When profiling, you want to ensure you focus your time and effort on areas where you can create the biggest impact. Thus it's recommended to start with a top-to-bottom approach when profiling, meaning you begin with a high-level overview of categories such as rendering, scripts, physics, and garbage collection (GC) allocations. Once you've identified areas of concern you can drill down into the deeper details. Use this high-level pass to collect data and take notes on the most critical performance issues, including scenarios that cause unwanted managed allocations or excessive CPU usage in your core game loop.

You'll need to first gather call stacks for GC.Alloc markers. If you're unfamiliar with this process, find some tips and tricks in the section titled Locating recurring memory allocations over application lifetime.

If the reported call stacks are not detailed enough to track down the source of the allocations or other slowdowns, you can perform a second profiling session with Deep Profiling enabled in order to find the source of the allocations. We cover deep profiling in more detail later in this guide but in summary, it's a mode in the Profiler that captures detailed performance data for every function call, providing granular insights into execution times and behaviors, but with significantly higher overhead compared to standard profiling.

When collecting notes on the frame time "offenders," be sure to note how they compare relative to the rest of the frame. This relative impact can be distorted when deep profiling is enabled, because deep profiling adds significant overhead by instrumenting every method call.

Profile early

While you should always profile throughout the entire development cycle of your project, the most significant gains from profiling are made when you start in the early phases.

Profile early and often so you and your team understand and memorize a "performance signature" for the project you can use to benchmark against. If performance takes a nosedive, you'll be able to easily spot when things go wrong and fix the issue.

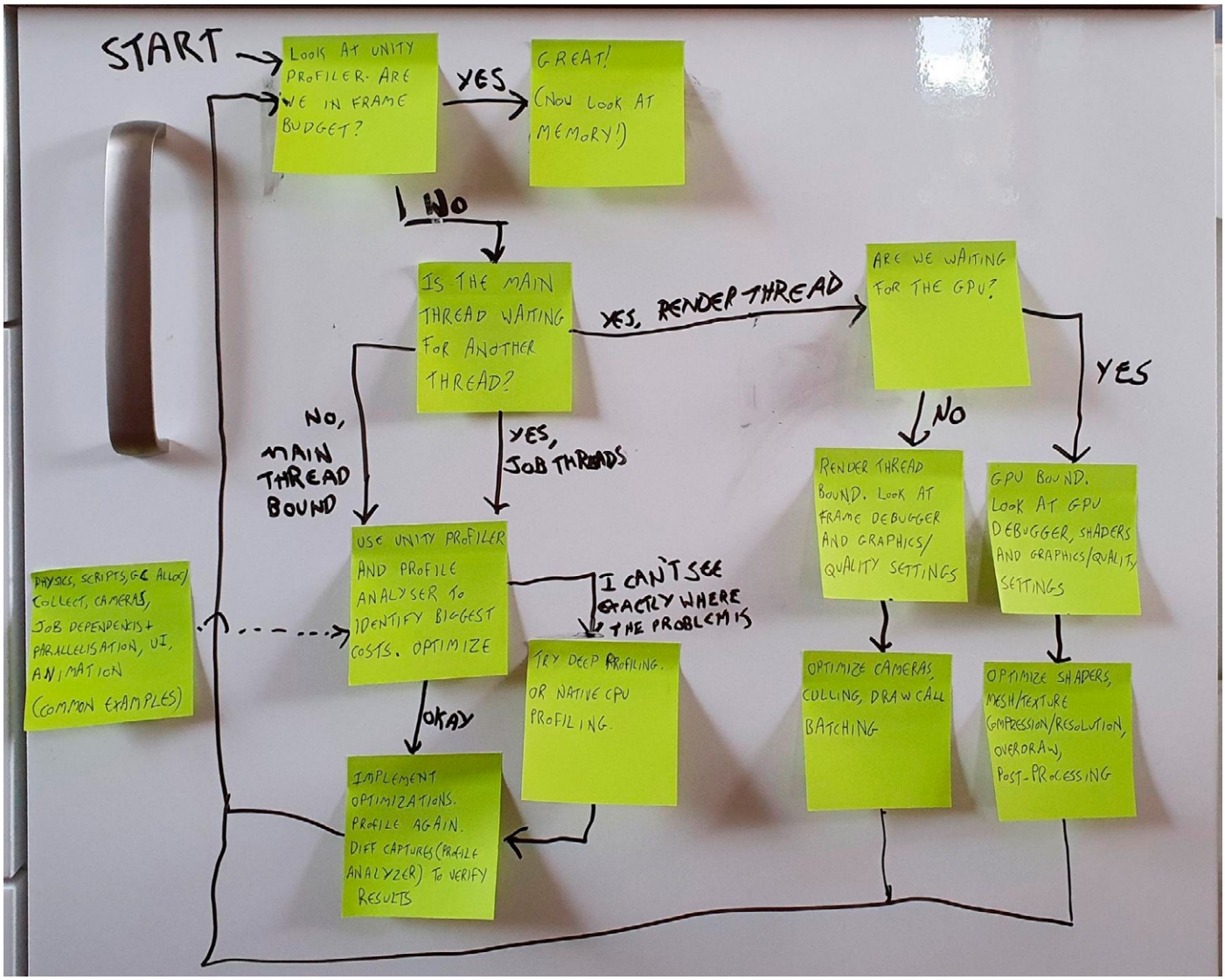
While profiling in the Editor gives you an easy way to identify the main issues, the most accurate profiling results always come from running and profiling builds on target devices, together with leveraging platform-specific tooling to dig into the hardware characteristics of each platform. This combination will provide you with a holistic view of application performance across all your target devices. For example, you might be GPU-bound on some mobile devices but CPU-bound on others, and you can only learn this by measuring on those devices.

Establish a profiling methodology

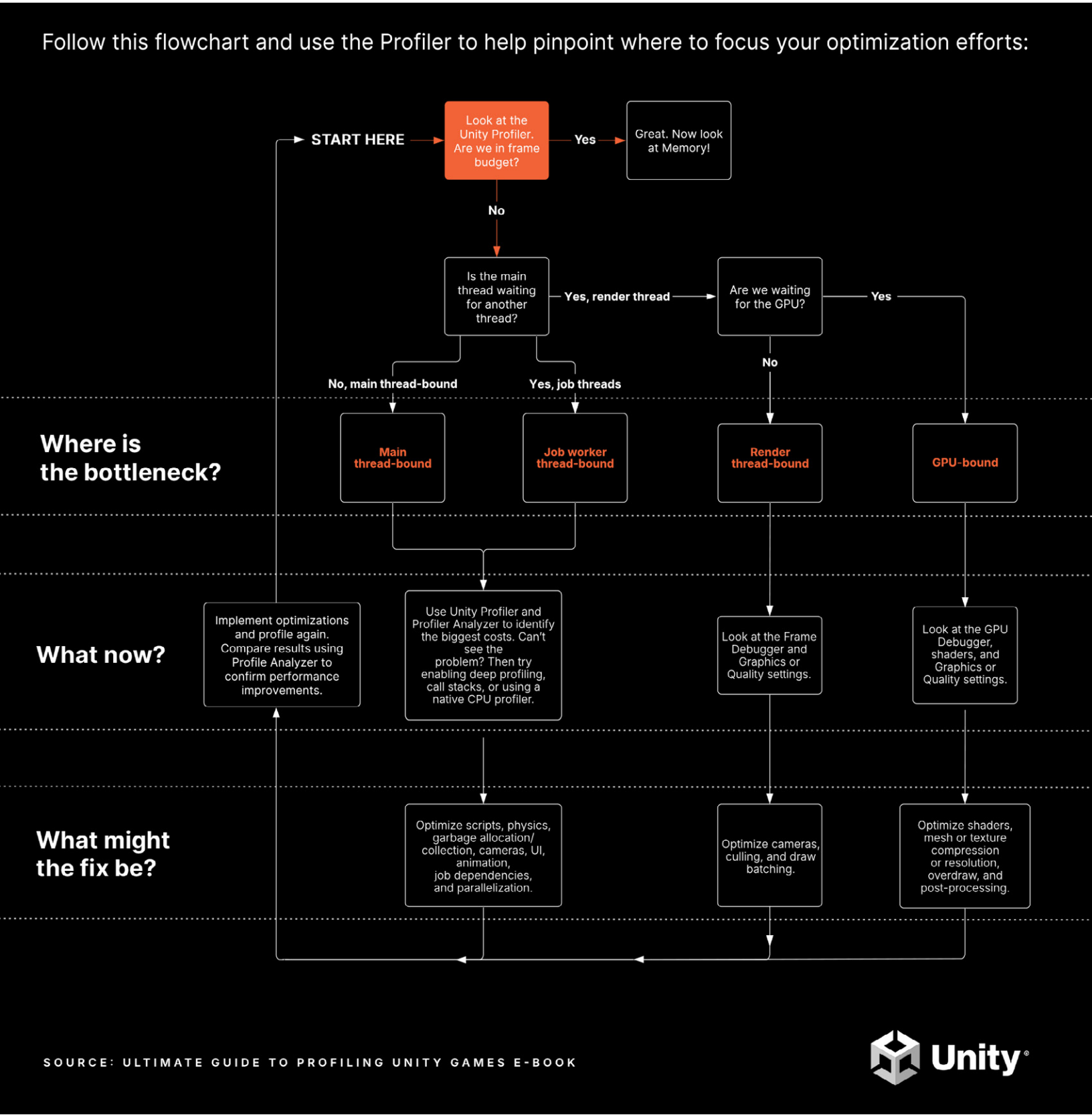
Profiling should be a structured process. Don't leave issues to be randomly caught. Instead it's better to establish a profiling methodology.

The point of profiling is to identify bottlenecks as targets for optimization. If you rely on guesswork, you can end up optimizing parts of the game that are not bottlenecks, resulting in little or no improvement to overall performance. Some "optimizations" might even worsen your game's overall performance while other ones can be labor-intensive but yield insignificant results. The key is to optimize the impact of your focused time investment.

The flow chart below illustrates the initial profiling process with the sections following it providing detailed information on each step. They also present Profiler captures from real Unity projects to illustrate the kinds of things to look for.



Follow this flowchart and use the Profiler to help pinpoint where to focus your optimization efforts:



Follow this flowchart and use the Profiler to help pinpoint where to focus your optimization efforts.

To get a holistic picture of all CPU activity, including when it's waiting for the GPU, use the **Timeline view** in the **CPU module** of the Profiler. Familiarize yourself with the common Profiler markers to interpret captures correctly. Some of the Profiler markers may appear differently depending on your target platform, so spend time exploring captures of your game on each of your target platforms to get a feel for what a "normal" capture looks like for your project.

A project's performance is bound by the chip and/or thread that takes the longest. That's the area on where optimization efforts should focus. For example, imagine the following scenarios for a game with a target frame time budget of 33.33 ms and VSync enabled:

— If the CPU frame time (excluding VSync) is 25 ms and GPU time is 20 ms, no problem! You're CPU-bound, but everything is within budget, and optimizing things won't improve the frame rate (unless you get both CPU and GPU below 16.66 ms and jump up to 60 fps).

— If the CPU frame time is 40 ms and GPU is 20 ms, you're CPU-bound and will need to optimize the CPU performance. Optimizing the GPU performance won't help; in fact, you might want to move some of the CPU work onto the GPU, for example, by using compute shaders instead of C# code where applicable, to balance things out.

— If the CPU frame time is 20 ms and GPU is 40 ms, you're GPU-bound and need to optimize the GPU work.

— If CPU and GPU are both at 40 ms, you're bound by both and will need to optimize both below 33.33 ms to reach 30 fps.

See these resources that further explore being CPU- or GPU-bound:

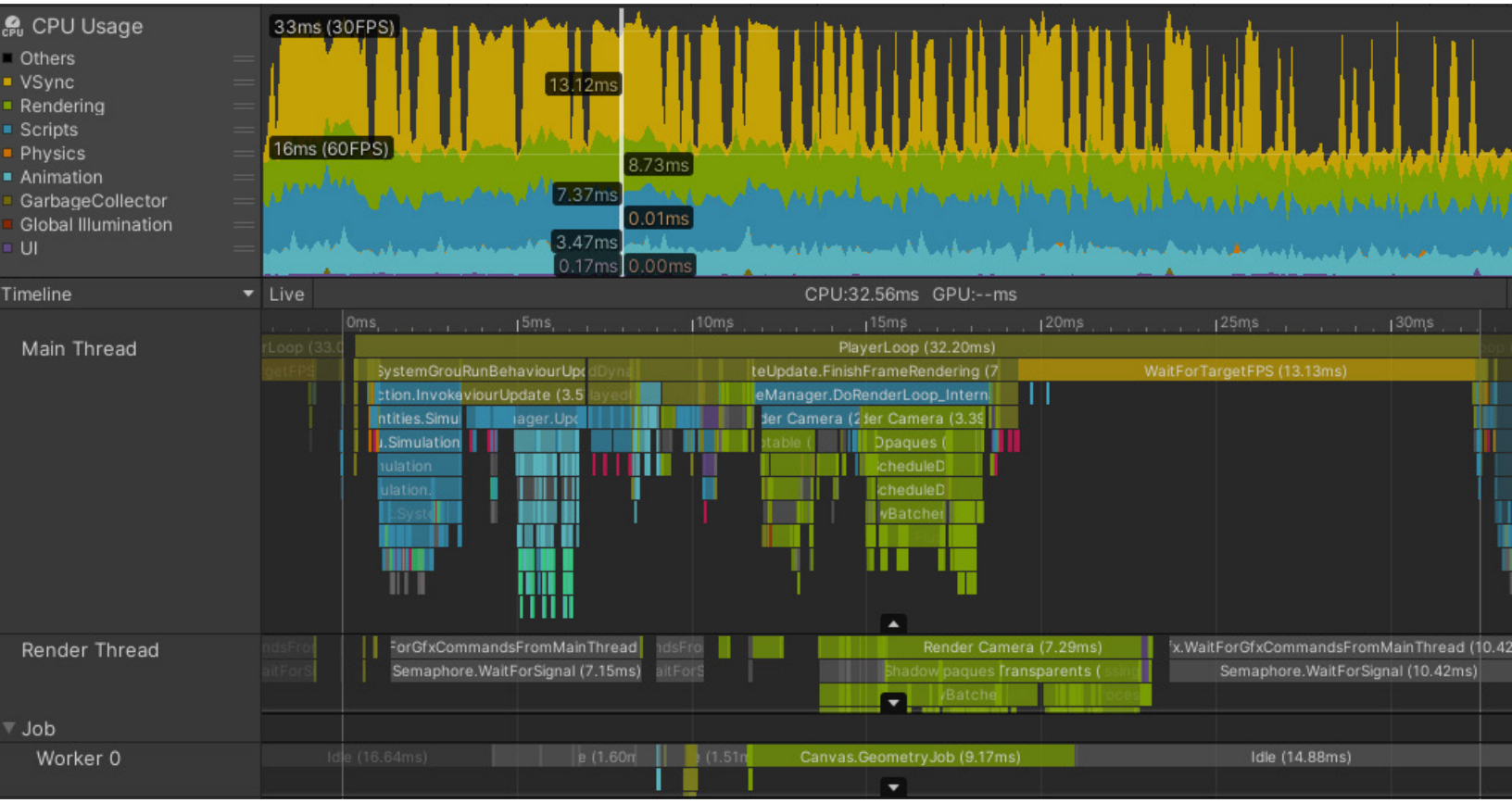
— Structure of a frame, the CPU and GPU

— Is your game draw call-bound?

Are you within frame budget?

Profiling and optimizing your project early and often throughout development will help you ensure that all of your application's CPU threads and the overall GPU frame time are within the frame budget. The question which will guide this process is, are you within the frame budget or not?

Below is an image of a profile capture from a Unity mobile game developed by a team that did ongoing profiling and optimization. The game targets 60 fps on high-spec mobile phones, and 30 fps on medium/low-spec phones, such as the one in this capture.



This is a profile of a game running comfortably within the ~22 ms frame budget required for 30 fps without overheating. Note the **WaitForTargetfps** padding the main thread time until VSync and the gray idle times in the render thread and worker thread. Also note that the VBlank interval can be observed by looking at the end times of **Gfx.Present** frame over frame, and that you can draw up a time scale in the Timeline view or on the Time ruler up top, to measure from one of these to the next.

Note how nearly half of the time on the selected frame is occupied by the yellow **WaitForTargetFPS** Profiler marker. The application has set **Application.targetFrameRate** to 30 fps, and **VSync** is enabled. The actual processing work on the main thread finishes at around the 19 ms mark, and the rest of the time is spent waiting for the remainder of the 33.33 ms to elapse before beginning the next frame. Although this time is represented with a Profiler marker, the main CPU thread is essentially idle during this time, allowing the CPU to cool and use a minimum of battery power.

The marker to look out for might be different on other platforms or if VSync is disabled. The important thing is to check whether the main thread is running within your frame budget or exactly on your frame budget with some kind of marker that indicates that the application is waiting for VSync and whether the other threads have any idle time.

Idle time is represented by gray or yellow Profiler markers. The screenshot above shows that the render thread is idling in **Gfx.WaitForGfxCommandsFromMainThread**, which indicates times when it has finished sending draw calls to the GPU on one frame, and is waiting for more draw call requests from the CPU on the next. Similarly, although the **Job Worker 0** thread spends some time in **Canvas.GeometryJob**, most of the time it's idle. These are all signs of an application that's comfortably within the frame budget.

If your game is in frame budget

If you are within the frame budget, including any adjustments made to the budget to account for battery usage and thermal throttling, you're finished with the key profiling tasks. You can conclude by running the **Memory Profiler** to ensure that the application is also within its memory budget.

CPU-bound

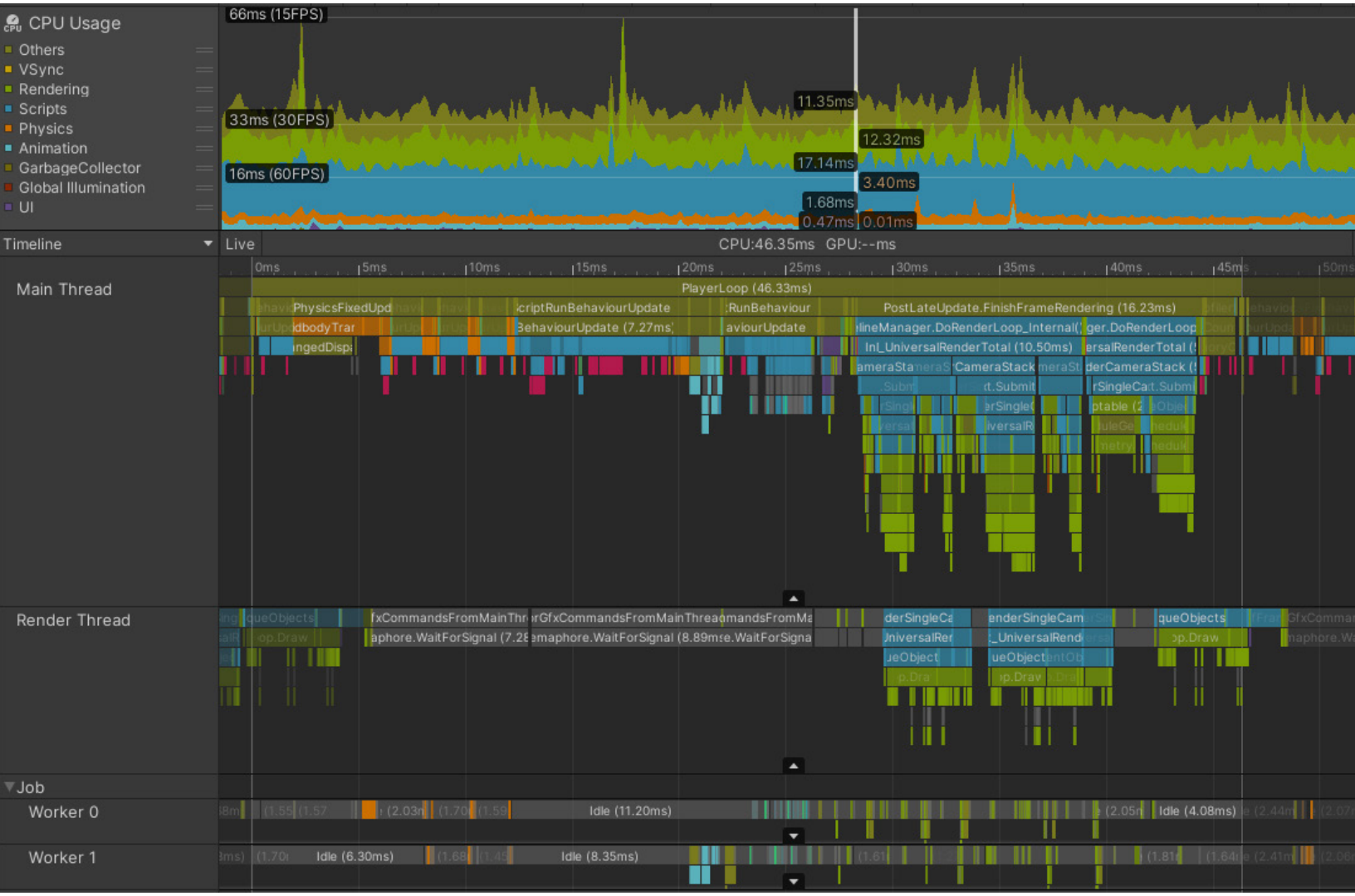
If your game is not within the CPU frame budget, the next step is to investigate what part of the CPU is the bottleneck – in other words, which thread is the most busy.

It's rare for the entire CPU workload to be the bottleneck. Modern CPUs have a number of different cores, capable of performing work independently and simultaneously. Different threads can run on each CPU core. A full Unity application uses a range of threads for different purposes, but those that are the most common for finding performance issues are:

- **The main thread**: This is where the majority of the game logic/scripts perform their work by default. Most Unity systems, such as physics, animation, UI, and the initial stages of rendering, execute here.
- **The render thread**: This handles the preparation work (e.g., which objects in the scene are visible to the camera and which are excluded/invisible because they're outside the view frustum, occluded, or culled by other criteria) that must happen before sending rendering instructions to the GPU.
    - During the rendering process, the main thread examines the scene and performs camera culling, depth sorting, and draw call batching, resulting in a list of things to render. This list is passed to the render thread, which translates it from Unity's internal platform-agnostic representation to the specific graphics API calls (like DirectX, Vulkan, or Metal) required to instruct the GPU on a particular platform.
- **The Job worker threads**: Developers can make use of the job system to schedule certain kinds of work to run on worker threads, which reduces the workload on the main thread. Some of Unity's systems and features also make use of the job system, such as physics, animation, and rendering.

A real-world example of main thread optimization

The image below shows how things might look in a project that is bound by the main thread. This project is running on a Meta Quest 2, which normally targets frame budgets of 13.88 ms (72 fps) or even 8.33 ms (120 fps), because high frame rates are important to avoid motion sickness in VR devices. However, even if this game was targeting 30 fps, it's clear that this project is in trouble.



Capture from a project which is main thread-bound

Although the render thread and worker threads look similar to the example which is within frame budget, the main thread is clearly busy with work during the whole frame. Even accounting for the small amount of profiler overhead at the end of the frame, the main thread is busy for over 45 ms, meaning that this project achieves frame rates of less than 22 fps. There is no marker that shows the main thread idly waiting for VSync; it's busy for the whole frame.

The next stage of investigation is to identify the parts of the frame that take the longest time and to understand why this is so. On this frame, **PostLateUpdate.FinishFrameRendering** takes 16.23 ms, more than the entire frame budget. Closer inspection reveals there are five instances of a marker called **Inl_RenderCameraStack**, indicating that five cameras are active and rendering the scene. Since every camera in Unity invokes the whole render pipeline, including culling, sorting, and batching, the highest-priority task for this project is reducing the number of active cameras, ideally to just one.

BehaviourUpdate, the Profiler marker that encompasses all `MonoBehaviour.Update()` method executions, takes 7.27 milliseconds in this frame.

In the Timeline view, magenta-colored sections indicate points where scripts are allocating managed heap memory. Switching to the **Hierarchy** view, and filtering by typing **GC.Alloc** in the search bar, shows that allocating this memory takes about 0.33 ms in this frame. However, that is an inaccurate measurement of the impact the memory allocations have on your CPU performance.

GC.Alloc markers are not timed by recording a Begin and End point like typical Profiler samples. To minimize their overhead, Unity records only the timestamp of the allocation and the allocated size.

The Profiler assigns a small, artificial sample duration to GC.Alloc markers solely to ensure they are visible in the Profiler views.

The actual allocation can take longer, especially if a new range of memory needs to be requested from the system. To see the impact more clearly, place Profiler markers around the code that does the allocation; in deep profiling, the gaps between the magenta-colored GC.Alloc samples in the Timeline view provide some indication of how long they might have taken.

Additionally, allocating new memory can have negative effects on performance that are harder to measure and attribute to them directly:

— Requesting new memory from the system might affect the power budget on a mobile device, which can lead to the system slowing down the CPU or GPU.

— The new memory likely needs to get loaded into the CPU's L1 Cache and thereby pushes out existing Cache lines.

— Incremental or synchronous garbage collection may be triggered directly or with a delay as the existing free space in Managed Memory is eventually exceeded.

At the start of the frame, four instances of **Physics.FixedUpdate** add up to 4.57 ms. Later on, **LateBehaviourUpdate** (calls to MonoBehaviour.LateUpdate()) take 4 ms, and **Animators** account for about 1 ms. To ensure this project hits its desired frame budget and rate, all of these main thread issues need to be investigated to find suitable optimizations.

Common pitfalls for main thread bottleneck

The biggest performance gains will be made by optimizing the things that take the longest time. The following areas are often fruitful places to look for optimizing in projects that are main thread-bound:

- Physics calculations
- MonoBehaviour script updates
- Garbage allocation and/or collection
- Camera culling and rendering on the main thread
- Inefficient draw call batching
- UI updates, layouts, and rebuilds
- Animation

Read our optimization guides that offer a long list of actionable tips for optimizing some of the most common pitfalls:

Get the e-book

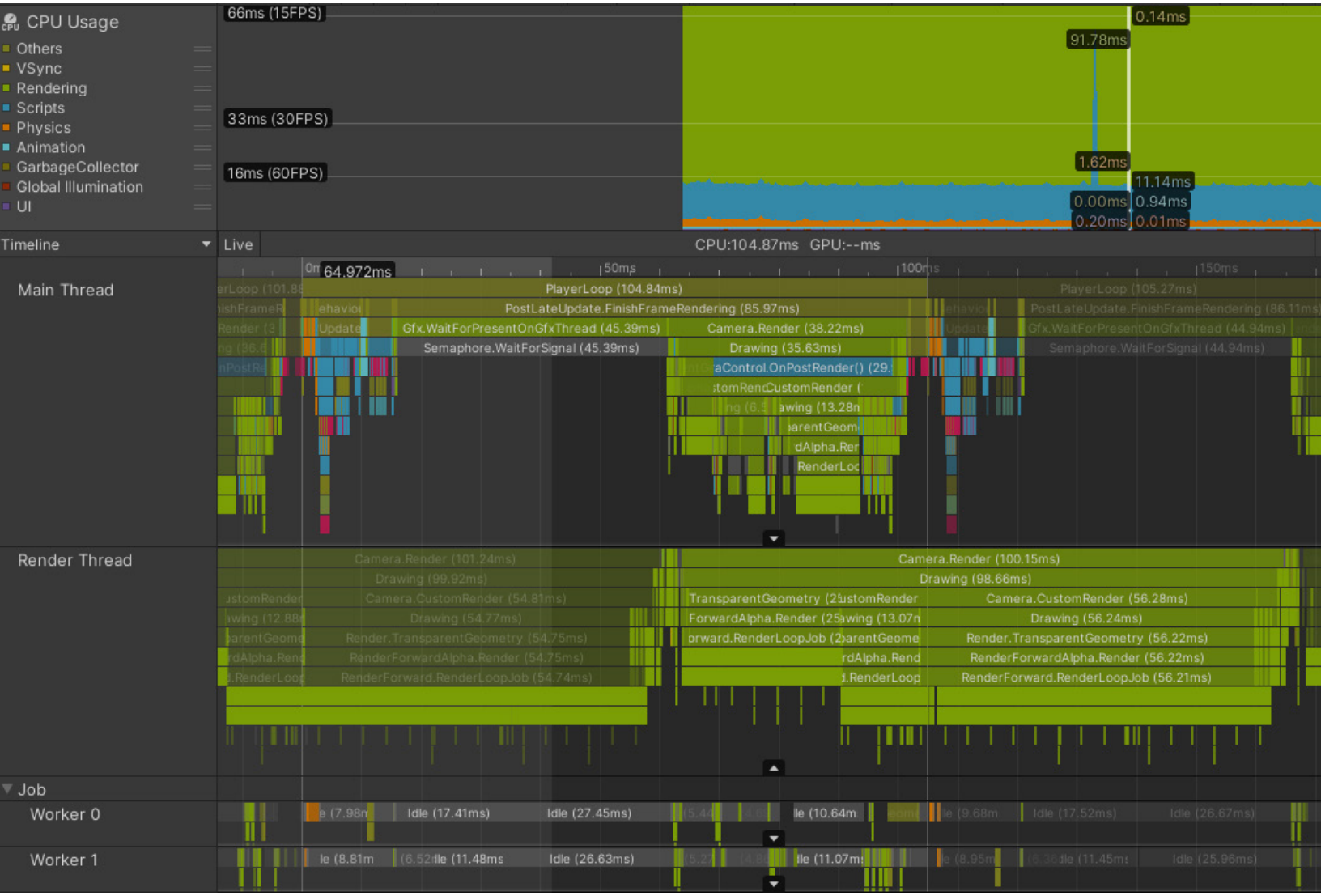
Get the e-book

Depending on the issue you want to investigate, other tools can also be helpful:

- For MonoBehaviour scripts that take a long time but don't show you exactly why that's the case, add Profiler Markers to the code or try deep profiling to see the full call stack.
- For scripts that allocate managed memory, enable allocation call stacks to see exactly where the allocations come from. Alternatively, enable deep profiling or use Project Auditor, which shows code issues filtered by memory, so you can identify all lines of code which result in managed allocations.
- Use the Frame Debugger to investigate the causes of poor draw call batching.

A real-world example of render thread optimization

Here's an actual project that's bound by its render thread. This is a console game with an isometric viewpoint and a target frame budget of 33.33 ms.



A Render thread-bound scenario

The Profiler capture shows that before rendering can begin on the current frame, the main thread waits for the render thread, as indicated by the **Gfx.WaitForPresentOnGfxThread** marker. The render thread is still submitting draw call commands from the previous frame and isn't ready to accept new draw calls from the main thread; it's also spending time in **Camera.Render**.

You can tell the difference between markers relating to the current frame and markers from other frames, because the latter appear darker. You can also see that once the main thread is able to continue and start issuing draw calls for the render thread to process, the render thread takes over 100 ms to process the current frame, which also creates a bottleneck during the next frame.

Further investigation showed that this game had a complex rendering setup, involving nine different cameras and many extra passes caused by replacement shaders. The game was also rendering over 130 point lights using a forward rendering path, which can add multiple additional transparent draw calls for each light. In total, these issues combined to create over 3000 draw calls per frame.

Common pitfalls for render thread bottlenecks

The following are common causes to investigate for projects that are render thread-bound:

— **Poor draw call batching**: This applies particularly on older graphics APIs such as OpenGL or DirectX 11.

— **Too many cameras**: Unless you're making a split-screen multiplayer game, the chances are that you should only ever have one active Camera.

— **Poor culling**: This results in too many things being drawn. Investigate your Camera's frustum dimensions and cull layer masks.

The Rendering Profiler module shows an overview of the number of draw call batches and SetPass calls every frame. The best tool for investigating which draw call batches your render thread is issuing to the GPU is the Frame Debugger.

Tools to solve the identified bottlenecks

While the focus of this e-book is about identifying performance issues, the two complementary performance optimization guides that we previously highlighted offer suggestions on how to solve the bottlenecks, depending on whether your target platform is PC or console or mobile. In the context of render thread bottlenecks it's worth emphasizing that Unity offers different batching systems and options depending on what problems you have identified. Here is a quick overview of some of the options which we explain in greater detail in the e-books:

— **SRP Batching** reduces CPU overhead by storing material data persistently in GPU memory. While it doesn't reduce actual draw call count, it makes each draw call cheaper.

— **GPU instancing** combines multiple instances of the same mesh using the same material into a single draw call.

— **Static Batching** combines static (non-moving) meshes sharing the same material and thus can give you wins when working with a level design with many static elements.

— **GPU resident drawer** automatically uses GPU instancing to reduce CPU overhead and draw calls, by grouping similar GameObjects together.

— **Dynamic Batching** combines small meshes at runtime which can be an advantage on older mobile devices with high draw call costs. However, the downside is that the vertex transformation can also be resource-intensive.

— **GPU occlusion culling** uses compute shaders to determine object visibility by comparing depth buffers from current and previous frames, reducing unnecessary rendering of occluded objects without requiring pre-baked data.

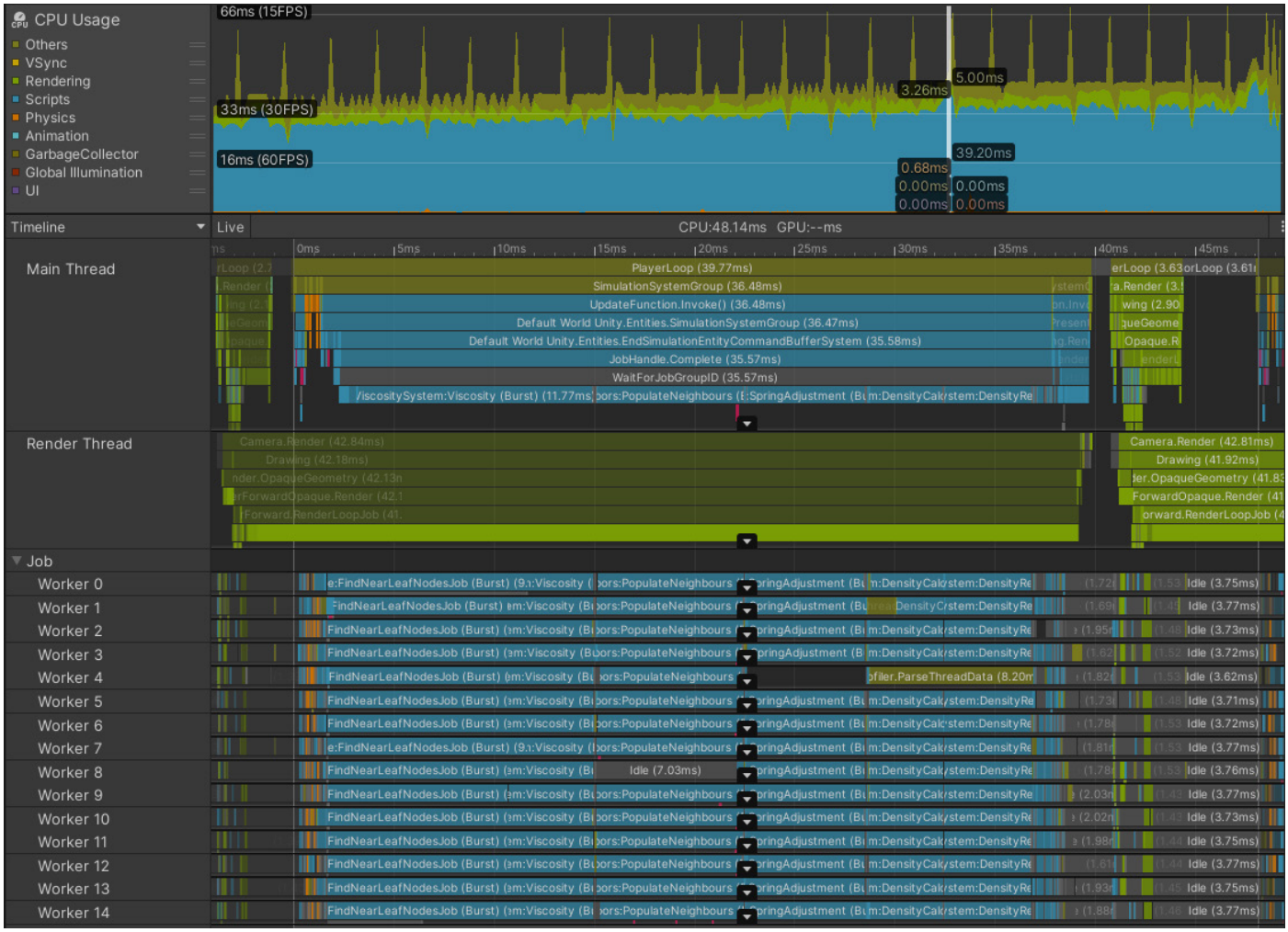
Additionally, on the CPU side, techniques such as **Camera.layerCullDistances** can be used to reduce the number of objects sent to the render thread by culling objects based on their distance from the camera, helping alleviate CPU bottlenecks during camera culling.

These are just some of the options available. Each one of these have different advantages and drawbacks. Some are limited to certain platforms. Projects need to often use a combination of several of these systems and to do so, an understanding of how to get the most out of them.

Worker threads

Projects bound by CPU threads other than the main or render threads are not that common. However, it can arise if your project uses the Data-Oriented Technology Stack (DOTS), especially if work is moved off the main thread into worker threads using the job system.

Here's a capture from Play mode in the Editor, showing a DOTS project running a particle fluid simulation on the CPU.



A DOTS-based project, heavy on simulation, bound by Worker threads

It looks like a success at first glance. The worker threads are packed tightly with Burst-compiled jobs, indicating a large amount of work has been moved off the main thread. Usually, this is a sound decision.

However, in this case, the frame time of 48.14 ms and the gray **WaitForJobGroupID** marker of 35.57 ms on the main thread, are signs that all is not well. WaitForJobGroupID indicates the main thread has scheduled jobs to run asynchronously on worker threads, but it needs the results of those jobs before the worker threads have finished running them. The blue Profiler markers beneath WaitForJobGroupID show the main thread running jobs while it waits, in an attempt to ensure the jobs finish sooner.

Although the jobs are Burst-compiled, they are still doing a lot of work. Perhaps the spatial query structure used by this project to quickly find which particles are close to each other should be optimized or swapped for a more efficient structure. Or, the spatial query jobs can be scheduled for the end of the frame rather than the start, with the results not required until the start of the next frame. Perhaps this project is trying to simulate too many particles. Further analysis of the jobs' code is required to find the solution, so adding finer-grained Profiler markers can help identify their slowest parts.

The jobs in your project might not be as parallelized as in this example. Perhaps you just have one long job running in a single worker thread. This is fine, so long as the time between the job being scheduled and the time it needs to be completed is long enough for the job to run. If it isn't, you will see the main thread stall as it waits for the job to complete, as in the screenshot above.

Common pitfalls for worker thread bottlenecks

Common causes of sync points and worker thread bottlenecks include:

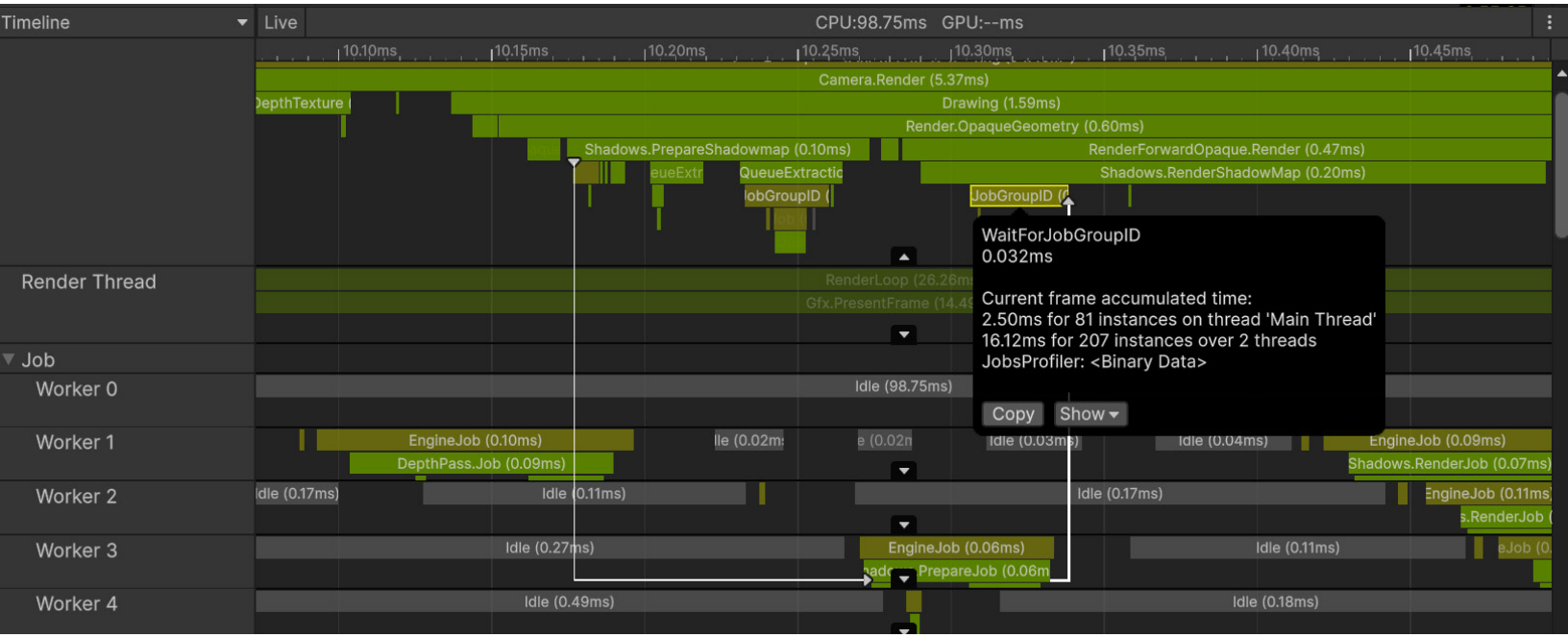
— Jobs not being compiled by the Burst compiler

— Long-running jobs on a single worker thread instead of being parallelized across multiple worker threads

— Insufficient time between the point in the frame when a job is scheduled and the point when the result is required

— Multiple "sync points" in a frame, which require all jobs to complete immediately

You can use the Flow Events feature in the Timeline view of the CPU Usage Profiler module to investigate when jobs are scheduled and when their results are expected by the main thread.



For more information about writing efficient DOTS code, see the DOTS Best Practices guide from Unity Learn.

GPU-bound

Your application is GPU-bound if the main thread spends a lot of time in Profiler markers like **Gfx.WaitForPresentOnGfxThread**, and your render thread simultaneously displays markers like **Gfx.PresentFrame** or **<GraphicsAPIName>.WaitForLastPresent.**

The best way of getting GPU frame times is using a target platform-specific GPU profiling tool, but not all devices make it easy to capture reliable data.

The FrameTimingManager API can be helpful in those cases, providing low-overhead, high-level frame times both on the CPU and GPU.

The following capture was taken on an Android mobile phone using the Vulkan graphics API. Although some of the time spent in Gfx.PresentFrame in this example might be related to waiting for VSync, the extreme length of this Profiler marker indicates the majority of this time is spent waiting for the GPU to finish rendering the previous frame.

A capture from a GPU-bound mobile game

In this game, certain gameplay events triggered the use of a shader that tripled the number of draw calls rendered by the GPU. Common issues to investigate when profiling GPU performance include:

- Expensive full-screen post-processing effects, like Ambient Occlusion and Bloom
- Expensive fragment shaders caused by:
  - Branching logic inside shader code
  - Using full float precision rather than half precision, especially on mobile
  - Excessive use of registers, which affect the wavefront occupancy of GPUs

- Overdraw in the Transparent render queue caused by:
  - Inefficient UI rendering
  - Overlapping or excessive use of particle systems
  - Post-processing effects
- Excessively high screen resolutions, such as:
  - 4K displays
  - Retina displays on mobile devices
- Micro triangles caused by:
  - Dense mesh geometry
  - Lack of Level of Detail (LOD) systems, which is a particular problem on mobile GPUs, but can affect PC and console GPUs as well
- Cache misses and wasted GPU memory bandwidth caused by:
  - Uncompressed textures
  - High-resolution textures without mipmaps
- Geometry or tessellation shaders, which may be run multiple times per frame if dynamic shadows are enabled

If your application appears to be GPU-bound you can use the Frame Debugger as a quick way to understand the draw call batches that are being sent to the GPU. However, this tool can't present any specific GPU timing information, only how the overall scene is constructed.

The best way to investigate the cause of GPU bottlenecks is to examine a GPU capture from a suitable GPU profiler. Which tool you use depends on the target hardware and the chosen graphics API. See the profiling and debugging tools section of this guide for more information.

Mobile challenges: Thermal control and battery lifetime

Thermal control is one of the most important areas to optimize for when developing applications for mobile devices. If the CPU or GPU spend too long working at full throttle due to inefficient code, those chips will get hot. To avoid overheating and potential damage to the chips the operating system will reduce the clock speed of the device to allow it to cool down, causing frame stuttering and a poor user experience. This performance reduction is known as thermal throttling.

Higher frame rates and increased code execution (or DRAM access operations) lead to increased battery drain and heat generation. Bad performance can also make your game unplayable for entire segments of lower-end mobile devices, which can lead to missed market opportunities.

When taking on the problem of thermals, consider the budget you have to work with as a system-wide budget.

Combat thermal throttling and battery drain by profiling early to optimize your game from the start. Dial in your project settings for your target platform hardware to combat thermal and battery drain problems.

Adjust frame budgets on mobile

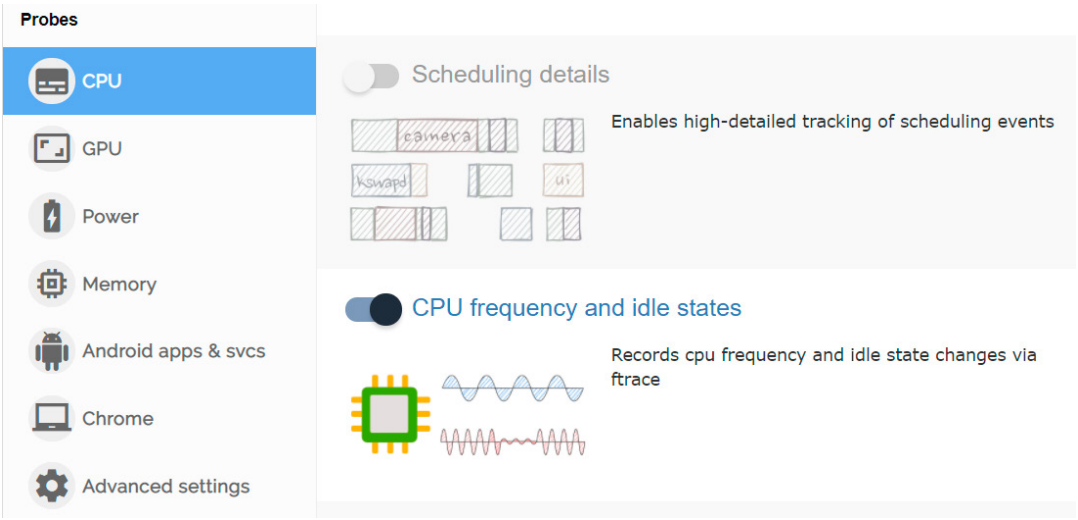
A general tip to combat device thermal issues over extended play times is to leave a frame idle time of around 35%. This gives mobile chips time to cool down and helps to prevent excessive battery drain. Using a target frame time of 33.33 ms per frame (for 30 fps), the frame budget for mobile devices will be approximately 22 ms per frame.

The calculation looks like this: **(1000 ms / 30) * 0.65 = 21.66 ms**

To achieve 60 fps on mobile using the same calculation would require a target frame time of (1000 ms / 60) * 0.65 = 10.83 ms. This is difficult to achieve on many mobile devices and would drain the battery twice as fast as targeting 30 fps. For these reasons, many mobile games target 30 fps rather than 60. Use Application.targetFrameRate to control this setting, and refer to the Set a frame budget section for more details about frame time.

Frequency scaling on mobile chips can make it tricky to identify your frame idle time budget allocations when profiling. Your improvements and optimizations can have a net positive effect, but the mobile device might be scaling frequency down, and as a result, running cooler. Use custom tooling such as FTrace or Perfetto to monitor mobile chip frequencies, idle time, and scaling before and after optimizations.

As long as you stay within your total frame time budget for your target fps (say 33.33 ms for 30 fps) and see your device working less or logging lower temperatures to maintain this frame rate, then you're on the right track.



Monitor CPU frequency and idle states with tools such as FTrace or Perfetto to help identify the results of frame budget allowance optimizations.

Another reason to add breathing room to frame budget on mobile devices is to account for real-world temperature fluctuations. On a hot day, a mobile device will heat up and have trouble dissipating heat, which can lead to thermal throttling and poor game performance. Set aside a percent of the frame budget to help avoid this scenario.

Reduce memory access operations

DRAM access is typically a power-hungry operation on mobile devices. Arm's optimization advice for graphics content on mobile devices says that LPDDR4 memory access costs approximately 100 picojoules per byte.

Reduce the number of memory access operations per frame by:

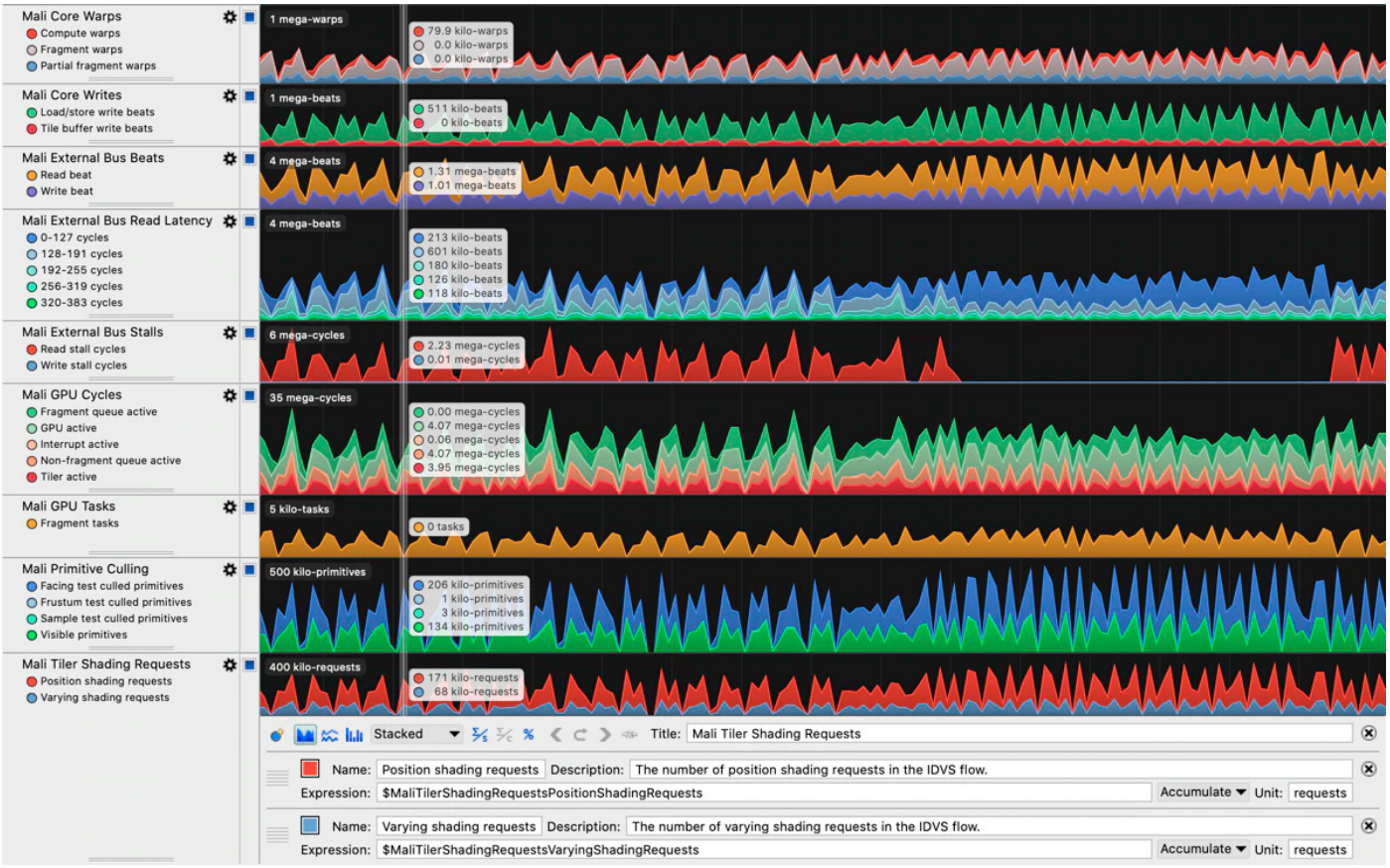
— Reducing frame rate

— Reducing display resolution where possible

— Using simpler meshes with reduced vertex count and attribute precision

— Using texture compression and mipmapping

When you need to focus on devices leveraging Arm CPU or GPU hardware, Arm Performance Studio tooling (specifically, Streamline Performance Analyzer) includes some great performance counters for identifying memory bandwidth issues. The available counters are listed and explained for each Arm GPU generation in a corresponding user guide, for example, Mali-G710 Performance Counter Reference Guide . Note that Arm Performance Studio GPU profiling requires an Arm Immortalis or Mali GPU.



Arm's Streamline Performance Analyzer includes a wealth of performance counter information that can be captured during live profiling sessions on target Arm hardware. This is great for identifying performance issues such as memory bandwidth saturation that result from overdraw.

A selected set of ARM hardware metrics is exposed to Unity Profiler and Players builds via Systemmetrics package.

Establish hardware tiers for benchmarking

In addition to using platform-specific profiling tools, establish tiers or a lowest-spec device for each platform and tier of quality you wish to support, then profile and optimize performance for each of these specifications.

As an example, if you're targeting mobile platforms, you might decide to support three tiers with quality controls that toggle features on or off based on the target hardware. You then optimize for the lowest device specification in each tier. As another example, if you're developing a game for consoles make sure you profile on both older and newer versions.

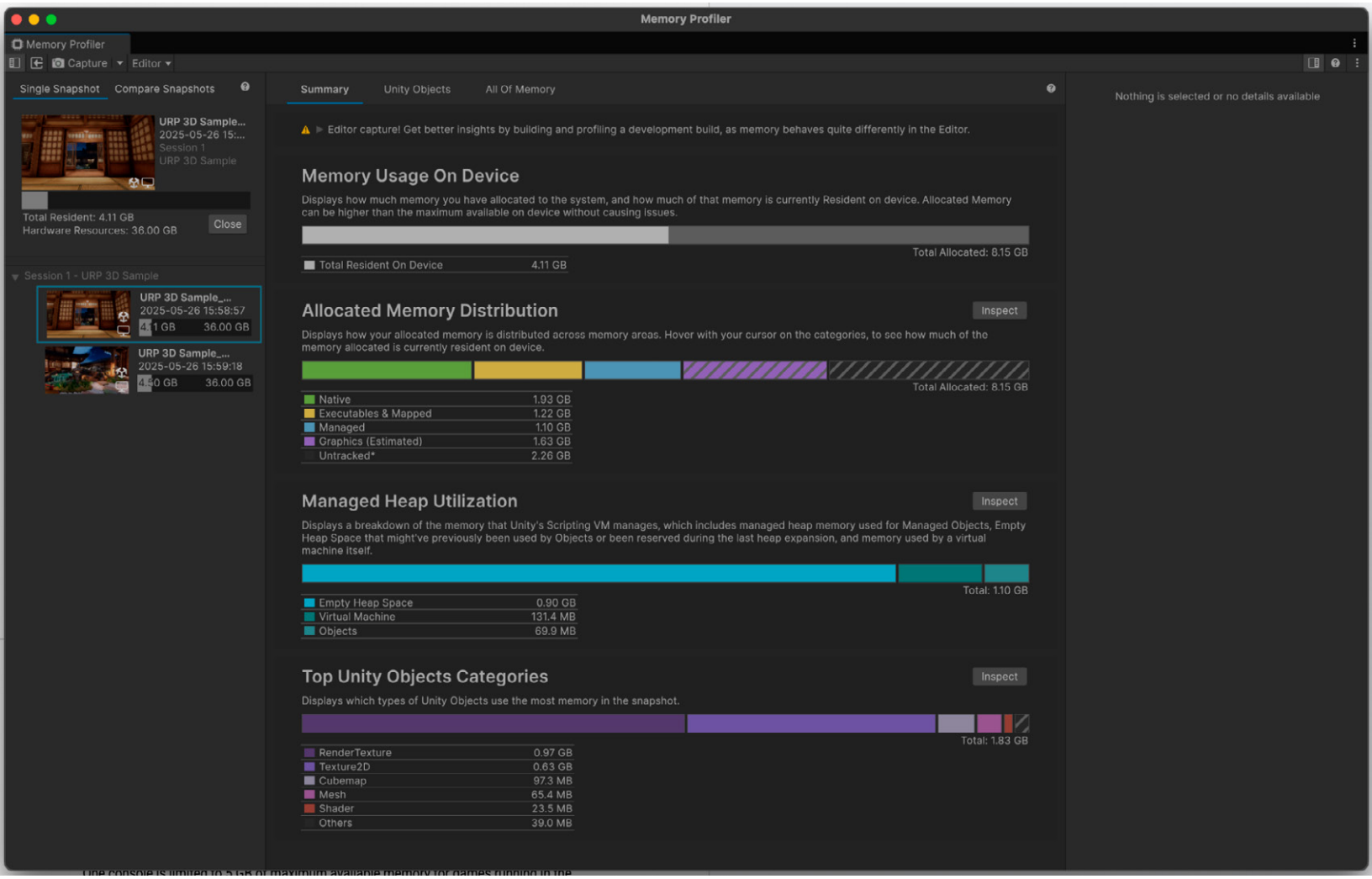
Our latest mobile optimization guide, (links to this guide and the PC/console optimization guide are in a previous section) has many tips and tricks that will help you reduce thermal throttling and increase battery life for mobile devices running your games.

Memory profiling

Memory profiling is largely unrelated to runtime performance but is useful for testing against hardware platform memory limitations or if your game is crashing. It can also be relevant if you want to improve CPU/GPU performance by making changes that actually increase memory usage.

There are two ways of analyzing memory usage in your application in Unity.

1. The Memory Profiler module: This is a built-in Profiler module that gives you basic information on where your application uses memory in the regular profiler.
2. The Memory Profiler: This is a dedicated tool available as a Unity package that you can add to your project. It adds an additional Memory Profiler window to the Unity Editor, which you can then use to analyze memory usage in your application in even more detail. You can store and compare snapshots to find memory leaks, or see the memory layout to find memory fragmentation issues. We will cover this in further detail later in this guide and keep focus here on the general considerations you need to take into account.



The Memory Profiler package is a tool you can use to inspect the memory usage of your Unity application and the Unity Editor.

Both these tools enable you to monitor memory usage, locate areas of an application where memory usage is higher than expected, and find and improve memory fragmentation.

Understand and define a memory budget

Understanding and budgeting for the memory limitations of your target devices are critical for multiplatform development. When designing scenes and levels, you need to stick to the memory budget that's set for each target device. By setting limits and guidelines, you can ensure that your application works well within the confines of each platform's hardware specification.

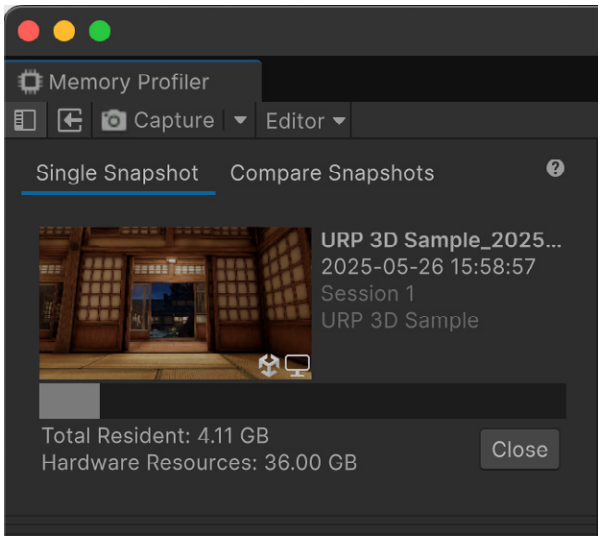
You can find device memory specifications in developer documentation.

It can also be useful to set content budgets around mesh and shader complexity, as well as for texture compression. These all play into how much memory is allocated. These budget figures can be referred to during the project's development cycle.

Determine physical RAM limits

As each platform has a memory limit, your application will need a memory budget for each of its target devices. Use the Memory Profiler to look at a captured snapshot of your memory usage. The **Hardware Resources** snapshot (see image below) shows **Physical Random Access Memory** (RAM) and **Video Random Access Memory** (VRAM) sizes. This figure doesn't account for the fact that not all of that space might be available to use. However, it provides a useful ballpark figure to start working with.

It's a good idea to cross reference hardware specifications for target platforms, as figures displayed here might not always show the full picture. Developer kit hardware sometimes has more memory, or you may be working with hardware that has a unified memory architecture.



The Hardware Resources snapshot shows the device RAM and VRAM figures the snapshot was captured on.

Determine the lowest specification to support for each target platform

Identify the hardware with the lowest specification of RAM for each platform you support, and use this to guide your memory budget decision. Remember that not all of that physical memory might be available to use. For example, a console could have a hypervisor running to support older games which might use some of the total memory. Think about a percentage (e.g., 80% of total) to use as a team depending on your specific scenario. For mobile platforms, you might also consider splitting into multiple tiers of specifications to support better quality and features for those with higher-end devices.

Consider per-team budgets for larger teams

Once you have a memory budget defined, consider setting memory budgets per team. For example, your environment artists get a certain amount of memory to use for each level or scene that is loaded, the audio team gets memory allocation for music and sound effects, and so on. While this may seem rigid, think of it as guidelines to inform the creative decisions being made vs the cost of the resources.

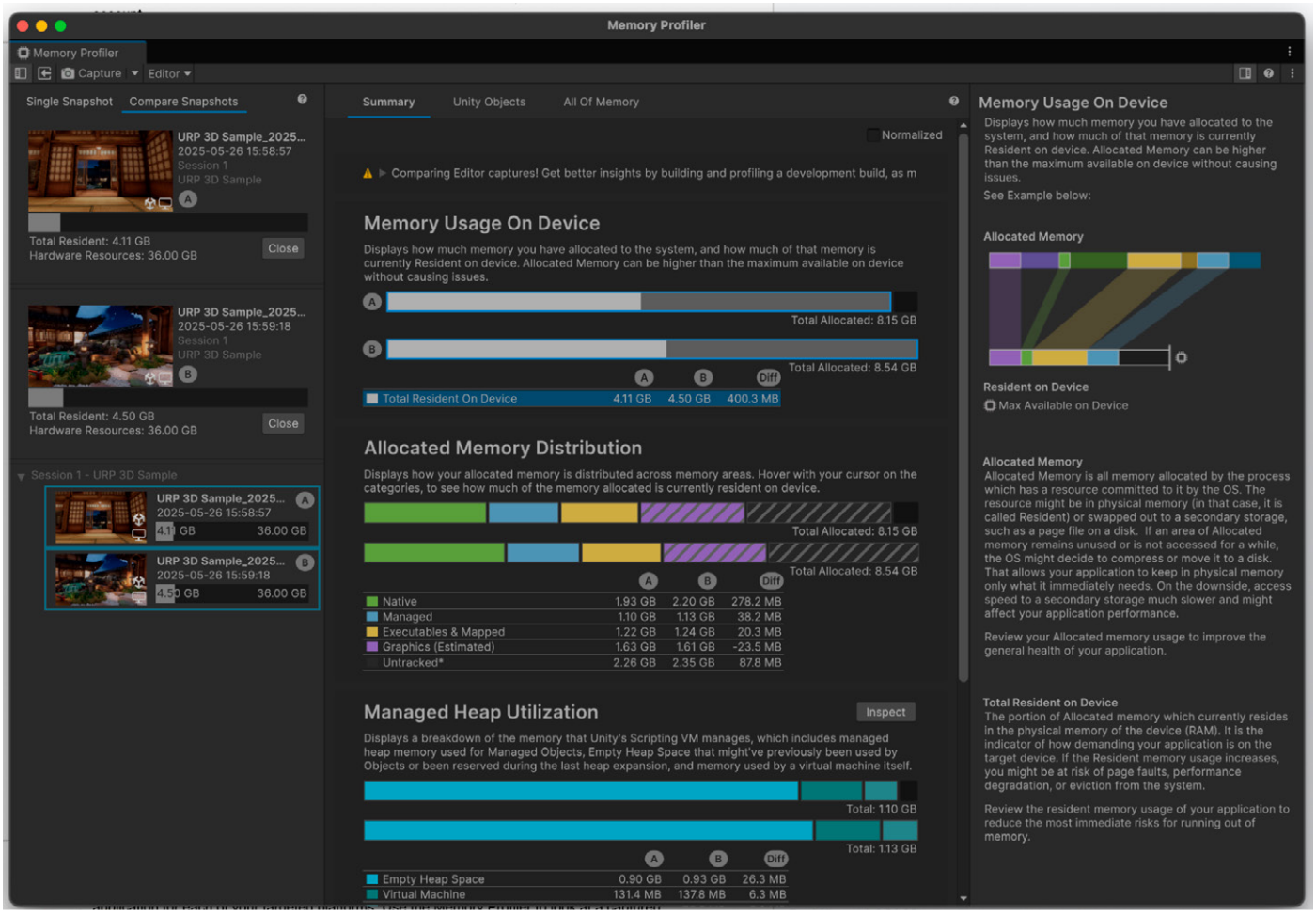
It's important to be flexible with the budgets as the project progresses. If one team comes in under budget, assign the surplus to another team if it can improve the areas of the game they're developing.

Once you decide on and set memory budgets for your target platforms, the next step is to use profiling tools to help you monitor and track memory usage in your game, enabling you to make informed decisions and take actions as needed.

In-depth analysis with the Memory Profiler package

The Memory Profiler package is useful for even more detailed memory analysis. Use it to store and compare snapshots to find memory leaks or see the memory layout of your application to find areas for optimization.

One great benefit of the Memory Profiler package is that, as well as capturing native objects (like the Memory Profiler module does), it also allows you to view Managed Memory, save and compare snapshots, and explore the memory contents via detailed, visual breakdowns of your memory usage.



The Memory Profiler package allows you to compare snapshots.

Read more about the Memory Profiler package in the Unity profiling and debug tools section.

A few tips to keep in mind when memory profiling

Remember to profile on the device that has the lowest specs for your overall target platform when setting a memory budget. Closely monitor memory usage, keeping your target limits in mind.

You'll usually want to profile using a powerful developer system with lots of memory available (space for storing large memory snapshots or loading and saving those snapshots quickly is important).

Memory profiling is a different beast compared with CPU and GPU profiling because it can incur additional memory overhead itself. You may need to profile memory on higher-end devices (with more memory), but specifically watch out for the memory budget limit for the lower-end target specification.

Settings such as quality levels, graphics tiers, and AssetBundle variants may have different memory usage on more powerful devices. With that in mind, here are some details to keep in mind to get the most from memory profiling:

— The quality and graphics settings can affect the size of render textures used for shadow maps.

— The resolution scaling can affect the size of the screen buffers, render textures, and post-processing effects.

— Texture settings can affect the size of all textures.

— The maximum LOD can affect models and more.

- If you have AssetBundle variants like an HD (High Definition) and an SD (Standard Definition) version, and you choose which one to use based on the specifications of your target device, you might get different asset sizes based on which device you are profiling on.
- The screen resolution of your target device will affect the size of render textures used for post-processing effects.
- The supported graphics API of a device might affect the size of shaders based on which variants of them it supports (or doesn't support).
- A tiered system that uses different quality and graphic settings, as well as AssetBundle variants, is a great way to be able to target a wider range of devices.
    - For example, you can load a HD version of an AssetBundle on a 4GB mobile device, and a SD version on a 2GB device. However, take the above variations in memory usage in mind and make sure to test both types of devices, as well as devices with different screen resolutions or supported graphics APIs.

Note: The Unity Editor will generally always show a larger memory footprint due to additional objects that are loaded from the Editor and Profiler. Additionally, texture memory footprint is higher since they are all forced to have read/write enabled in the Editor.

Unity profiling and debug tools

Unity offers a suite of tools that help you prevent, identify, and fix performance problems. We mentioned several of these throughout the guide so far. Now let's take a closer look at when to use which one.

Some of the tooling mentioned in this section falls under static analyzers or the debugging tools category, for example, the Frame Debugger. While they're not profilers, they're important to include in your toolkit when it comes to analyzing and improving your Unity projects.

What are the differences between profiling, debugging, and static analysis tools?

Profiling tools instrument and collect timing data relating to code execution.

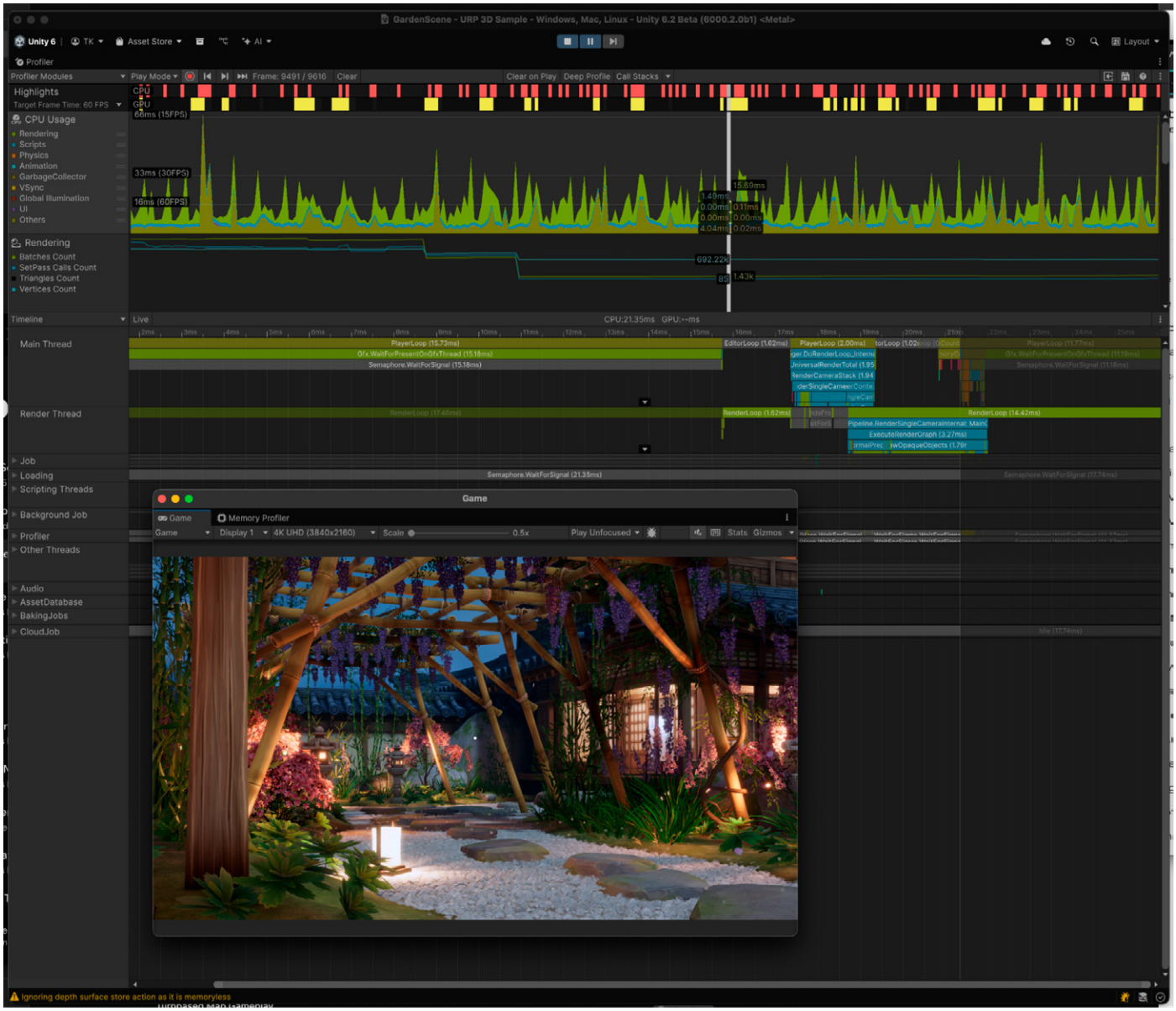
Debugging tools allow you to step through the execution of a program, pause and examine values, and provide many other advanced features. For example, the Frame Debugger lets you step through the rendering of frames, examine shader values, and more.

Static analyzers are programs that can take source code or other assets as input and analyze them using built-in rules to reason about the "correctness" of said input, without needing to run the project.

Unity Profiler

The built-in Unity Profiler helps you detect the causes of any bottlenecks or freezes at runtime and better understand what's happening at a specific frame or point in time. As explained earlier in this guide, the Profiler allows you to both profile builds of your application as well as profile

directly in the Editor. However, it adds some overhead and can skew your results. Also, keep in mind that your development machine is probably more powerful than your target device.



The Unity Profiler in action, profiling the garden scene from the URP 3D Sample

The Profiler includes a Deep Profile setting, which is helpful when you need detailed insights into the specific code being executed at runtime.

Additionally, the Unity Profiler enables comparison across different modules, allowing you to focus on specific parts of your application. We recommend that you always enable the **CPU**, **Memory**, and **Renderer** modules for a comprehensive view of your application's performance. Enable other modules as you need them, like Audio or Physics, based on the type of issues you are investigating.

Get started with profiling in Unity

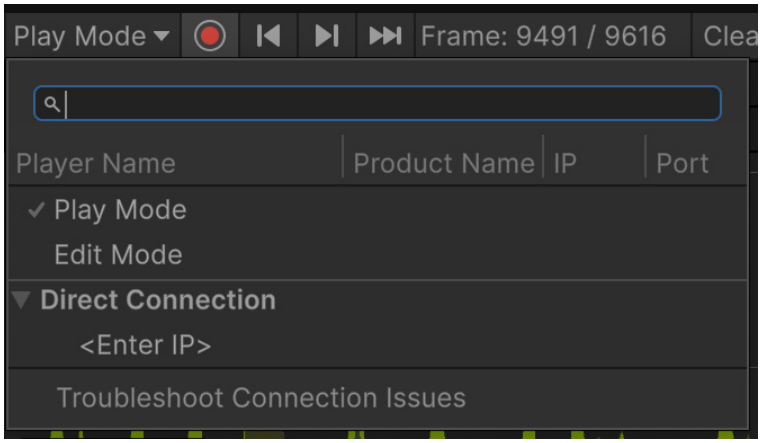
If you are new to the Unity Profiler check out our video tutorial here.

Alternatively you can try following these steps to get started:

- You must use a development build when profiling. Do this via **File > Build Profiles** and then check the **Development Build** checkbox.
- Tick the **Autoconnect Profiler** checkbox (this is optional).
    - Note: Autoconnect Profiler can add up to 10 seconds to initial startup time and should only be enabled if you want to profile your first scene's initialization. If you don't enable Autoconnect Profiler, you can always connect the Profiler to a running development build manually.
- Build for the target platform.
- Open the Unity Profiler via **Window > Analysis > Profiler**.
- Disable any Profiler modules you don't need. Each enabled module incurs a performance overhead for the player (you can observe some of this overhead using the Profiler. CollectGlobalStats marker).
- Set the **Frame Count** option in the **Preferences > Profiler** window. A higher number here will give you more frames you can analyze in the Profiler window, at the expense of using some extra memory on your Editor machine.
- Disable your device mobile network, and leave WiFi enabled.
- Run the build on your target device.
    - If you select Autoconnect Profiler, then the build will have the Editor machine's IP address baked in. At launch, the application will attempt to connect directly to the Unity Profiler at this IP address. The Profiler will automatically connect and begin displaying frame and profiling information.
    - If you did not select Autoconnect Profiler, then you will need to manually connect to your Player using the **Target Selection** dropdown.

The new Highlights module in the top bar makes it easier to identify where your project struggles to meet the target frame rate.

To save on build time (at the cost of reduced accuracy), profile your application running directly in the Unity Editor. Choose Play mode from the **Attach to Player** dropdown menu in the Profiler window.



Using the Profiler to target the game running in Play mode

Unity Profiler tips

Disable the VSync and Others markers in the CPU Profiler module

The **VSync** marker represents “dead time,” wherein the CPU main thread is idle while waiting for VSync. Hiding markers can sometimes make it difficult to understand how other category times came to be, or even how the total frame time is formed. With this in mind, another option is to reorder the list so that VSync is at the top. This provides a clearer view of the graph where the “noise” added by the VSync marker is reduced and the overall picture clearer.

The **Others** marker represents profiling overhead and can be safely ignored since it won’t be present in final builds of your project.

Disable VSync in the build

Another option for getting a clear picture of how the main thread, render thread, and GPU are interacting is to profile a build in which VSync is disabled entirely. To do this:

1. Go to **Edit > Project Settings...**
2. Select **Quality** and click on the Quality Level(s) to be used on your target device.
3. Set **VSync Count** to **Don’t Sync**.
4. Make a Development build of the game and connect it to the Profiler.

Instead of waiting for the next VBlank, the game will begin a frame as soon as the previous frame is complete. Disabling VSync can cause visual artifacts, such as tearing, on some platforms (in which case, remember to re-enable it for release builds), but removing the artificial wait can make profiler captures easier to read, particularly when you’re investigating where the bottlenecks are in your project.

Know when to profile in Play mode or Editor mode

When using the Profiler, you can choose Play mode, Editor, or a remote or attached device as the Player target.

Use Play mode to profile your game/application, and Editor mode to see what the Unity Editor surrounding the game is doing.

Using the Editor as the target for profiling has a high impact on profiling accuracy. The Profiler window is effectively profiling itself recursively. However, it can be valuable to profile the Editor if its performance slows down. You can then identify scripts and extensions that are slowing the Editor down and hampering productivity.

Examples of when you might want to profile the Editor include:

— If it takes a long time to enter Play mode after pressing the Play button

— If the Editor becomes sluggish or unresponsive

— If a project takes a long time to open

The blog post "Tips for working more effectively with the Asset Database" describes how to use the **-profiler-enable** command line option to start profiling from the moment the Editor starts running.

Use Standalone Profiler

Use the Standalone Profiler to launch the Profiler in its own dedicated process, separate from the Unity Editor, when you want to perform Play mode or Editor profiling. This avoids the Profiler UI or Editor from having an effect on measured timings. You'll also get a cleaner set of profiling data to filter and work with.

Starting the Profiler as a standalone process

Profile in the Editor for quick iterations

Profile in the Editor when you want to quickly iterate on fixing performance issues. For example, if a performance problem is spotted in the build, profile in the Editor to verify that you can also find it there. If you do find the problem, use Play mode profiling to quickly iterate on changes toward a potential solution. Once the issue is solved, make a build and verify the solution also works on target devices.

This workflow is optimal because you spend less time building changes and deploying to devices. Instead, you can iterate quickly in the Editor and use profiling tools to validate your change results.

Using the Memory Profiler module

Many of the features of the Memory Profiler module are superseded by the Memory Profiler package, but you can still use the module to supplement your memory analysis efforts.

Use the **Detailed** view in the Memory Profiler module to drill down into the highest memory trees to find out what is using the most memory.

The Memory Profiler module allows you to easily see how much memory you have allocated to the system.

Here are some more resources to help you explore additional use cases and features of the Unity Profiler:

— Profiler overview in the Unity manual

— Introduction to profiling in Unity

— How to profile and optimize a game

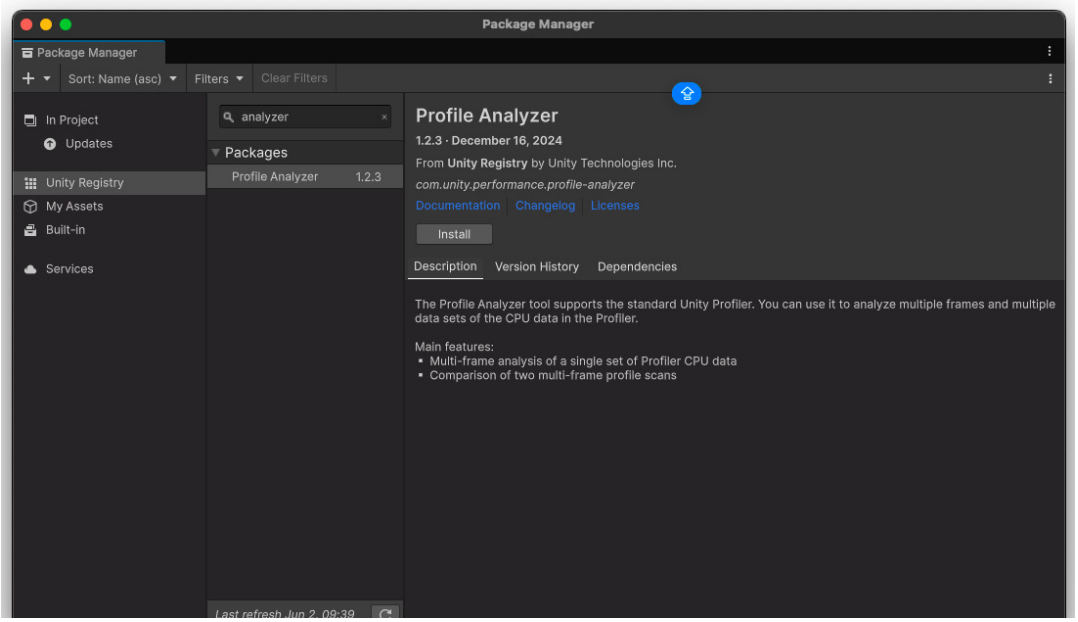
— Unity Profiler Walkthrough & Tutorial

Profile Analyzer

While the standard Unity Profiler enables detailed analysis of individual frames, the Profile Analyzer aggregates and visualizes marker data captured from multiple Unity Profiler frames, providing a broader, "'big picture" overview. This makes it easy to compare and analyze performance data across multiple frames or across different profiling sessions.

To get started with the Profile Analyzer:

1. Install the Profile Analyze Package via **Window > Package Management > Package Manager**.
2. Go to the Unity Registry and browse or use the search filter to find the Profile Analyzer package.

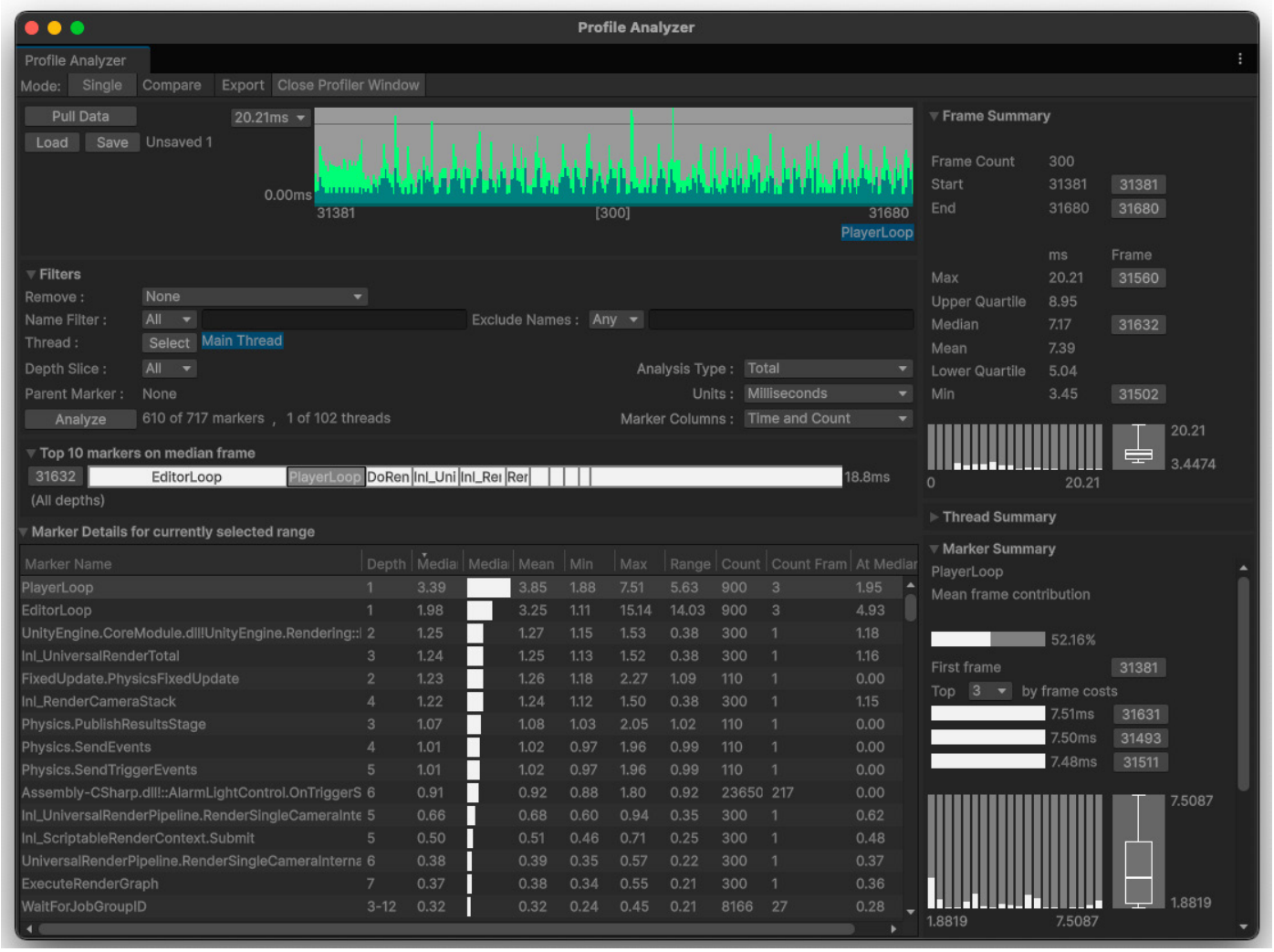


Install the Profile Analyzer from the Package Manager.

The Profile Analyzer pulls a set of frames captured in the Unity Profiler and performs statistical analysis on them. The data it displays provides useful performance timing information for each function, such as Min, Max, Mean, and Median timings.

As the Profile Analyzer is great for performing comparisons of data sets, consider using it throughout your game development to get clarity on performance and optimization challenges. You can also use it to A/B test a game scenario for performance differences, compare before and after profiling data for code refactoring and optimization, new features, or even Unity version upgrades.

One useful tip is to save profiling sessions to compare before and after performance optimization work when using the Profile Analyzer.



A great companion to the Unity Profiler, the Profile Analyzer aggregates and compares multiple frames captured in profiling sessions. This is a screenshot of the Single view.

To start, you first need to capture data using the Profiler and then populate the Profile Analyzer with that data to perform an analysis.

Using aggregated data gives you a more informed way of looking at what's going on in your game, rather than viewing only one frame at a time. For example, in a 300-frame (10-second) gameplay capture or a 20-second loading sequence you might need to know:

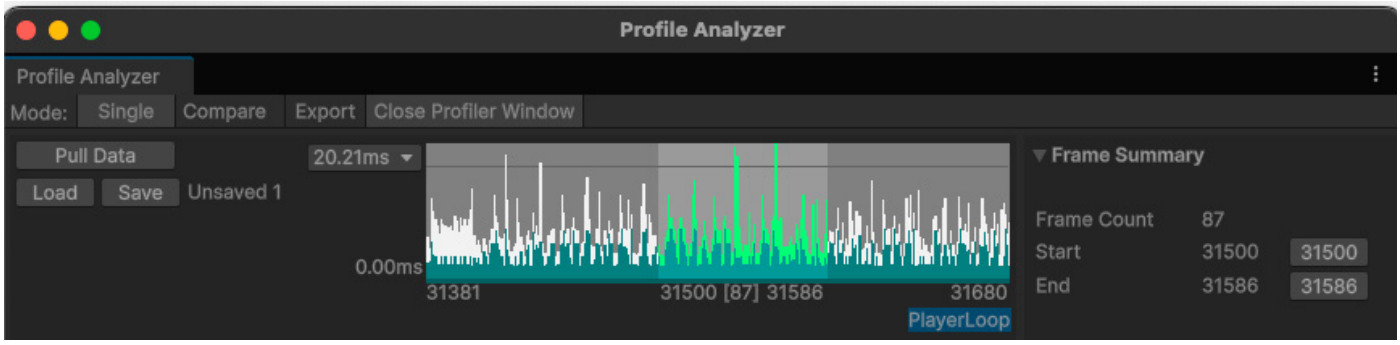
— What are the biggest CPU costs on the main and render threads?

— What is the mean/median/total cost of each of those markers?

Answering these essential questions can help you locate the biggest problems and prioritize their optimizations.

The statistics and detail available with Profile Analyzer allow you to delve deeper into the performance characteristics of your code when running across multiple frames, or even compared with previous profile capture sessions.

Use the Frame Control panel to select one, or a range, of frames. When selected, the Marker Details pane updates to show aggregated data for the selection with a sortable list of markers containing useful statistics.



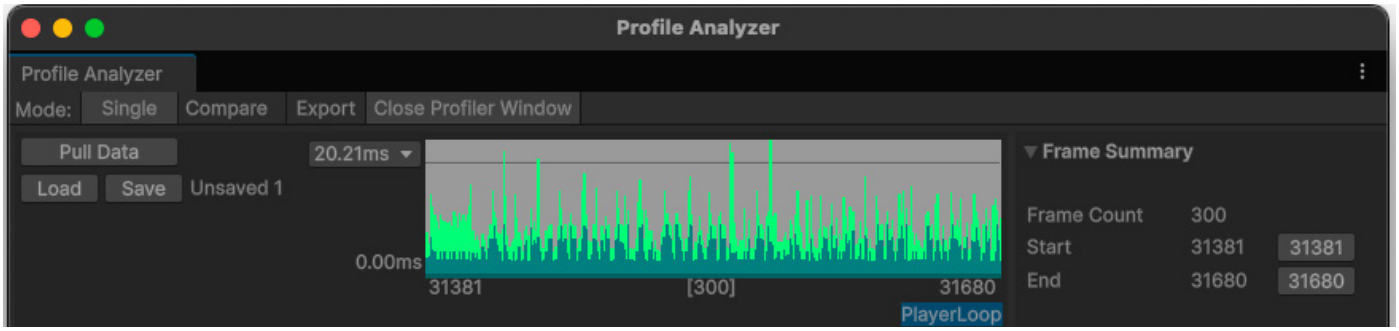
Use the Frame Control Panel to select the range of frames you would like to focus on.

The Marker Summary pane displays in-depth information on selected markers. Each marker in the list is an aggregation of all the instances of that marker, across all filtered threads in the range of selected frames.

The Marker Summary panel contains detailed information about each marker aggregation selected in the Marker Details panel.

Profile Analyzer views

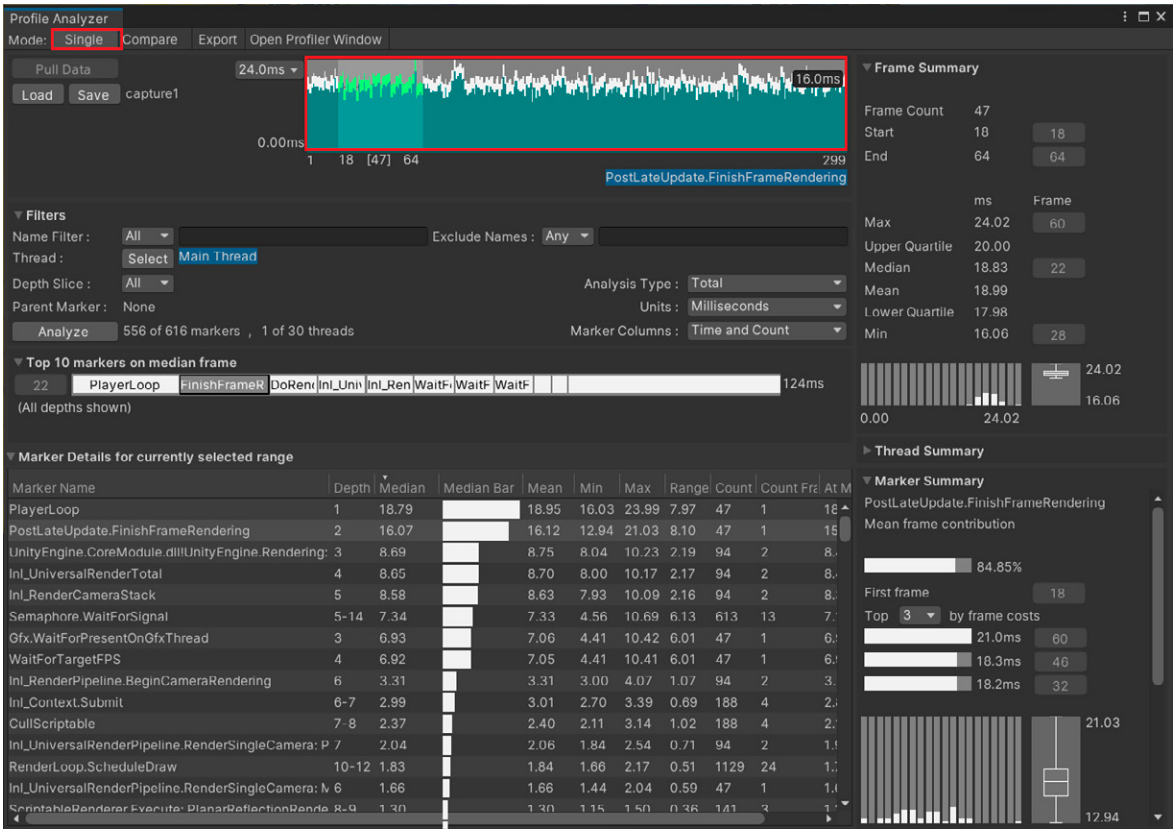
Notice the **Mode** selection in the top of the window. The Profile Analyzer has multiple views and approaches for analyzing profiling data. Use the different views to select, sort, view, and compare sets of profiling data.



You can select between different modes in the top of the panel.

Single view

The **Single view** is the default starting point of the Profile Analyzer, providing answers to high-level performance-over-time questions up front. The Single view displays information about a single set of captured profile data. Use it to analyze how profile markers perform across frames. This view is divided into several panels, which contain information on timings, as well as min, max, median, mean, and lower/upper quartile values for frames, threads, and markers.

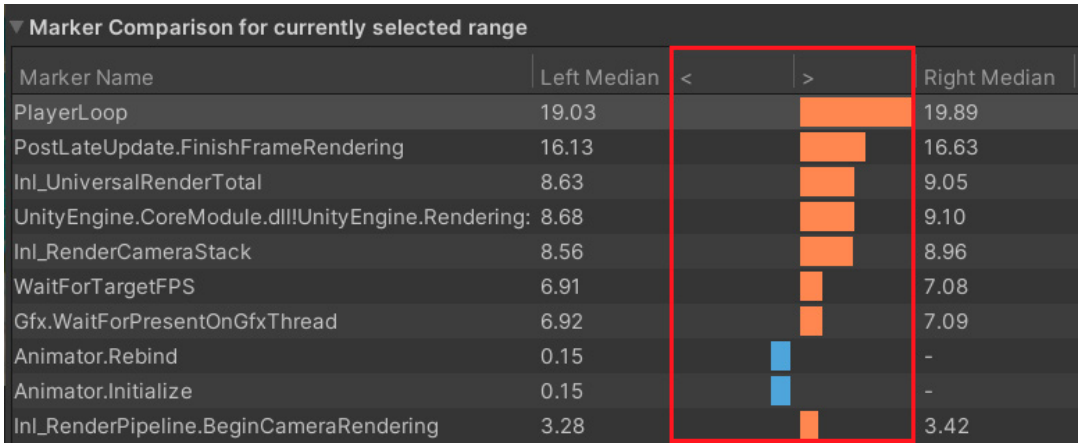


The Single view shows profile marker statistics and timings for a single or range of frames.

The many useful insights it can provide makes the Single view an essential part of any profiling toolkit.

Compare view

The Compare view is particularly effective for analyzing performance variations, as it allows you to load two distinct data sets which are then displayed in different colors for clear, side-by-side comparison.



Data set marker timings can be easily compared in the Compare view using the Marker Comparison pane and its color coding.

Use the following steps to compare performance changes using the Profile Analyzer. You can either use the **Pull Data** option from an active Unity Profiler capture or the **Load Data** option from a saved session. When loading, files must be in the Profile Analyzer's .pdata format. For Unity Profiler .data files, open them first in the Profiler window, then use Pull Data in the Profile Analyzer. It's also recommended to save your original .data files from the Profiler.

1. **Prepare a test:** Choose a consistent section of your game to profile for a meaningful benchmark comparison. A scripted or repeatable manual playthrough works best so you minimize random side effects that impact performance.
2. **Capture "before" data**:
    - Open Profile Analyzer (**Window > Analysis > Profile Analyzer**).
    - In the Unity Profiler, record a profiling session of your chosen gameplay before making any optimizations.
    - In the Analyzer's **Compare** tab, click the first **Pull Data** button. This loads the current capture from the Profiler or, alternatively, you can save the session.
3. **Optimize and capture "after" data**:
    - Apply your code or performance improvements.
    - Clear the Unity Profiler's previous data, then record a new profiling session of the same gameplay.
    - In Profile Analyzer, click the second Pull Data button to load this new session.

4. **Analyze differences**:

    — The **Marker Comparison** pane shows how marker timings differ between your “before” (left) and “after” (right) captures.

    — Columns marked with **<** or **>** indicate which capture had a larger value for that metric.

    — You can change which metrics are compared using the **Marker Columns** filter.

Refer to the Compare view entry page for more details on each Marker Comparison column.

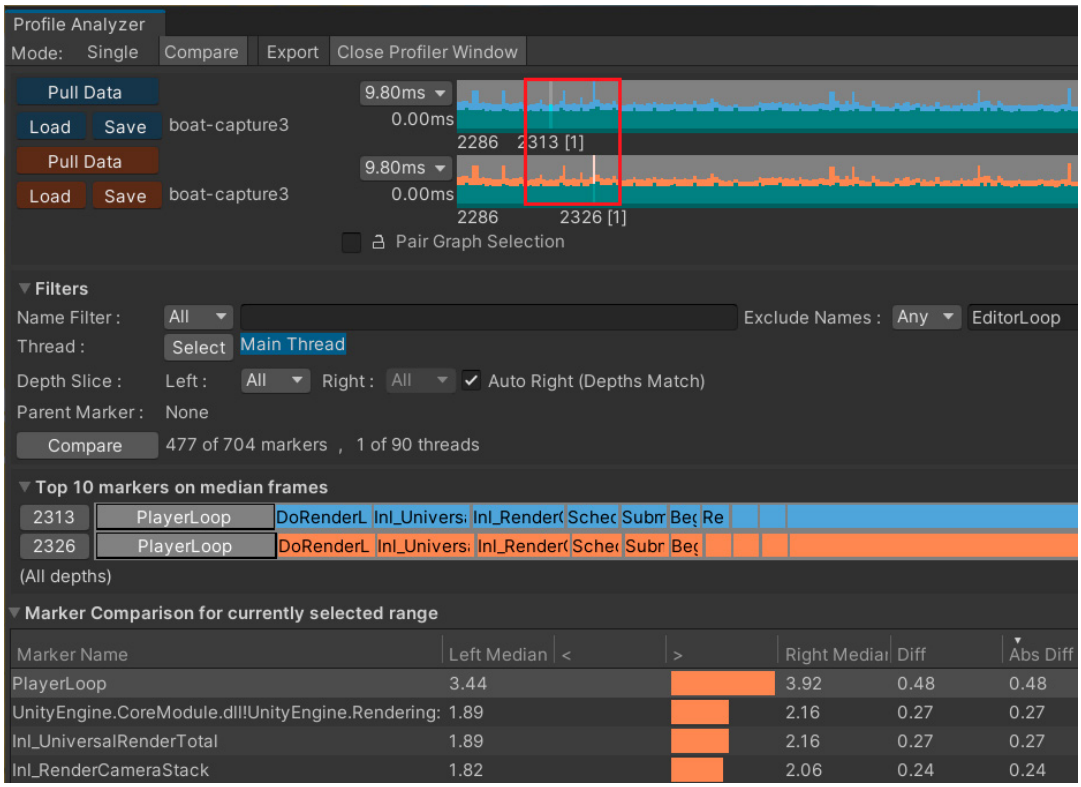
Comparing median and longest frames

Compare the median and longest frames within a single Profiler capture to pinpoint things happening in the latter that do not appear in the former, or to see what is taking longer than average to complete.

Open the Profile Analyzer Compare view and load the same data set for both the left and right sides. You can also load a data set in the Single view, then switch to Compare.

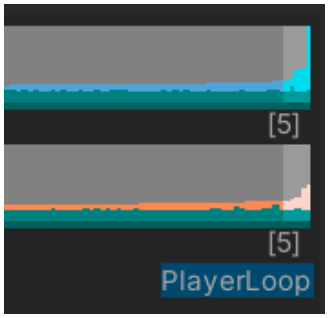
Right-click the top **Frame Control** graph, and choose **Select Median Frame**. Right-click the bottom graph, and choose **Select Longest Frame**.

The Profile Analyzer Marker Comparison panel updates to display the differences.



Comparing the median and longest frames from a capture

Another useful trick for comparing data is to sort both graphs by frame duration (**Right-click > Order By Frame Duration**), then select a range in each set, either focusing on, or excluding, the outlier frames (frames that are disproportionately long or short).



Ordering Frames by Duration and selecting an outlier range

This lets you compare the most typical frames against the most extreme ones. The data is then displayed in the Marker Comparison table for the selected range, making it easier to analyze what contributes to performance spikes or inconsistencies.

Learn more about the Profiler Analyzer with these resources:

— Profile Analyzer Walkthrough & Tutorial

— CPU performance analysis with Unity's Profile Analyzer

— Introduction to profiling

Profile Analyzer tips

— Drill into user scripts (ignoring Unity Engine API levels) by selecting a **Depth level** of **4**. After filtering to this level and looking at the Unity Profiler in **Timeline mode**, you can correlate the call stack depth to make a selection here – Monobehaviour scripts will appear in blue on the fourth level down. This is a quick way to see if your specific logic and gameplay scripts are taxing by themselves without any other "noise."

— Filter data in the same way for other areas of the Unity engine, such as animators or engine physics.

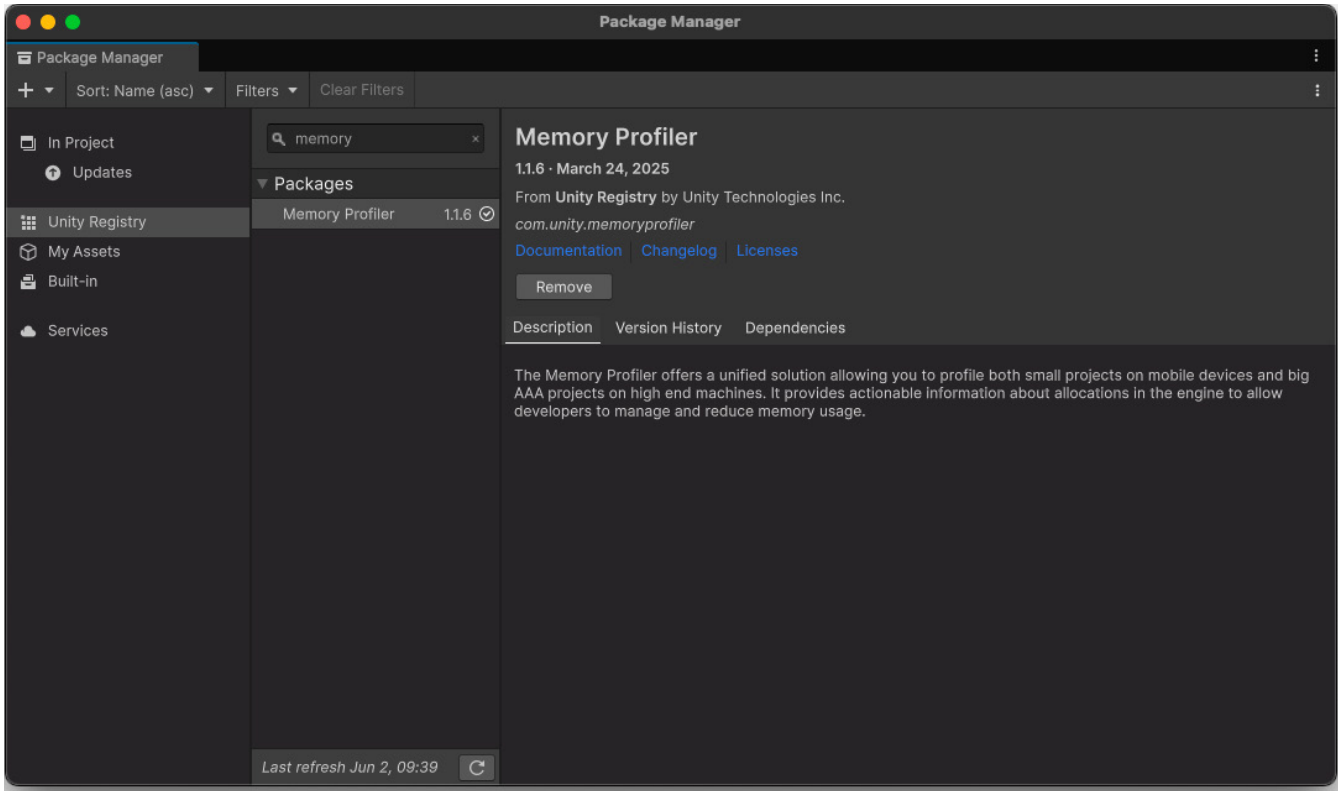
— On the right side in the **Frame Summary** section, you'll find the highlighted method's performance range histogram. Hover over the **Max Fram**e number (the exact frame in which max timing was found) to get a clickable link to view the frame selection in the Unity Profiler. Use this view to analyze other factors that potentially contribute to the high maximum frame time.

— If you have a widescreen or two monitors available it can be useful to open the Profile Analyzer and the Unity Profiler side by side. This setup enables you to double-click a frame in the Profile Analyzer to automatically select the same frame in the Unity Profiler, from where you can further investigate it using the Timeline or Hierarchy views.

Memory Profiler

The Memory Profiler package can help you understand and optimize your project's memory usage. It allows you to capture 'snapshots' of your application's memory at specific moments, both within the Unity Editor and in running Player builds on your target device.

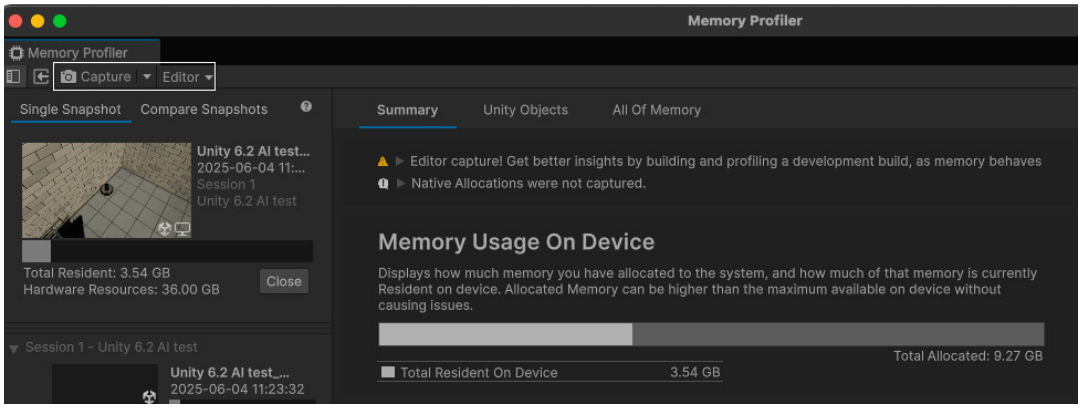
The snapshots provide a comprehensive breakdown of how memory is being utilized, showing allocations throughout the engine. This helps you identify sources of excessive or unnecessary memory usage, track down memory leaks, and inspect issues like heap fragmentation.



The Memory Profiler is a package available in Package Manager.

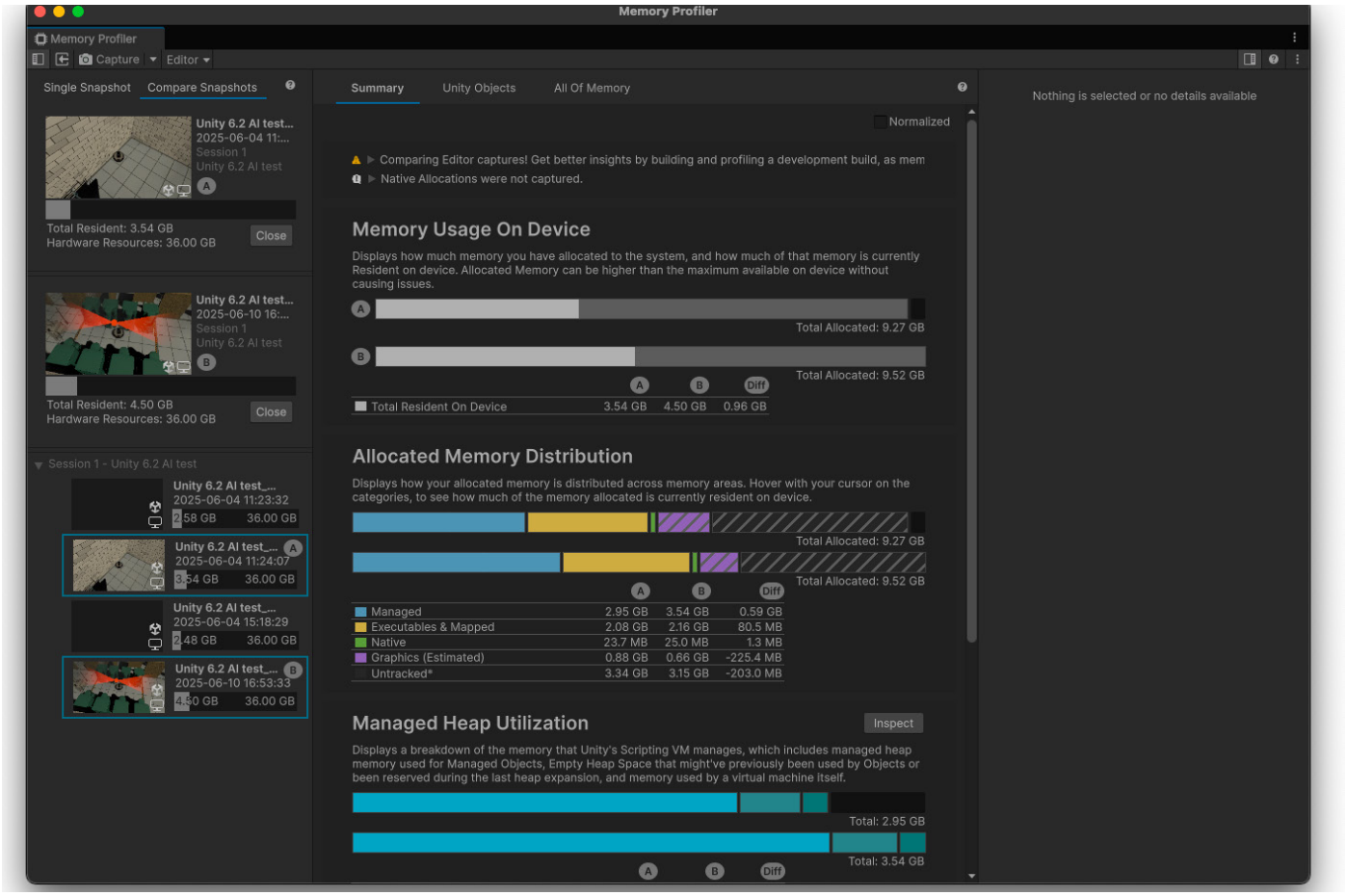
After installing the Memory Profiler package, open it via **Window > Analysis > Memory Profiler**.

The Memory Profiler's top menu bar allows you to change the player selection target and capture or import snapshots. The Target selection dropdown in the top-left corner (in the image below, the target chosen is "Editor") allows you to profile memory directly on your target hardware by connecting the Memory Profiler to the remote device. Note that profiling in the Unity Editor will give you inaccurate figures due to overheads added by the Editor and other tooling.



Change player selection and capture or import memory snapshots.

On the left side of the Memory Profiler window is the Snapshots component. Use this to manage and open or close saved memory snapshots. The Snapshot component provides two views, Single Snapshot and Compare Snapshot.



You can manage multiple memory snapshots.

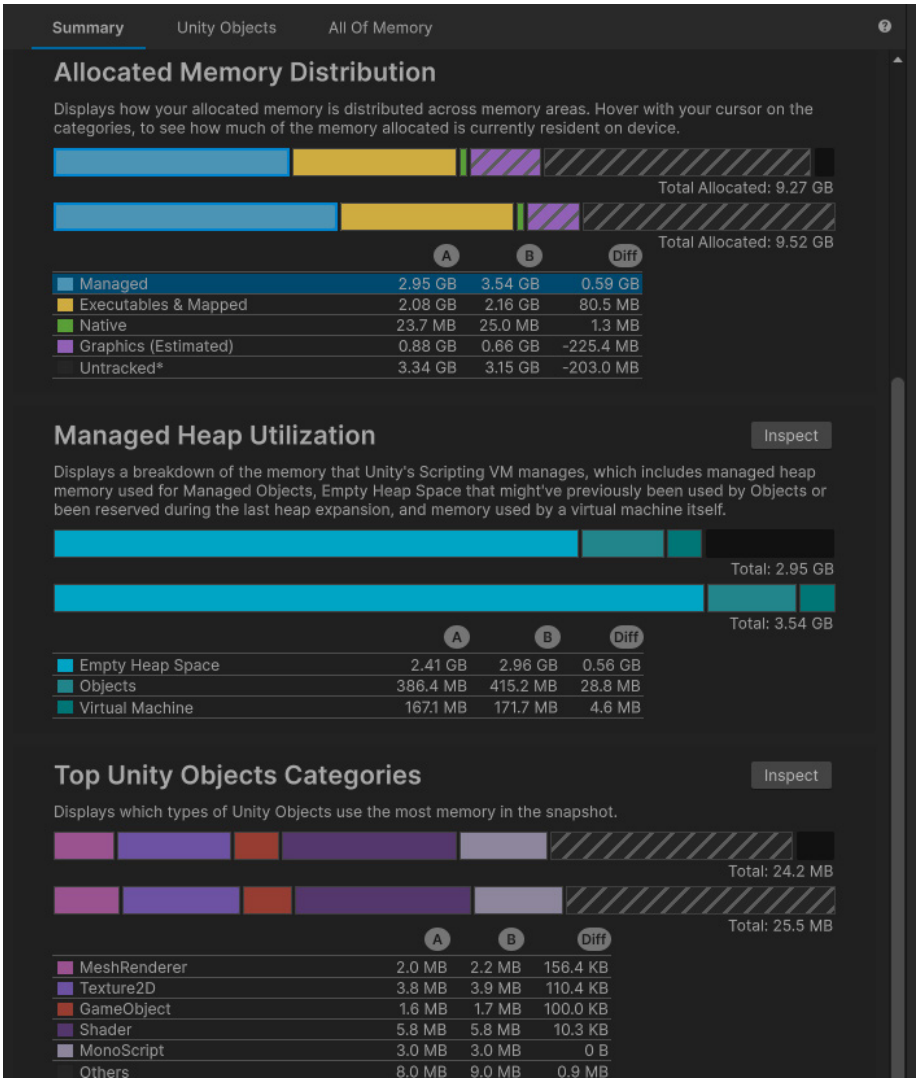
Similar to the Profile Analyzer, the Memory Profiler allows you to load and compare two memory snapshots side by side. Use this comparison to track memory growth over time, analyze usage between scenes, or identify potential memory leaks.

Memory Profiler has a number of tabs in the main window that allow you to dig into memory snapshots, the key ones being **Summary**, **Unity Objects**, and **All of Memory**. Let's look at each of these options in detail.

The Summary tab

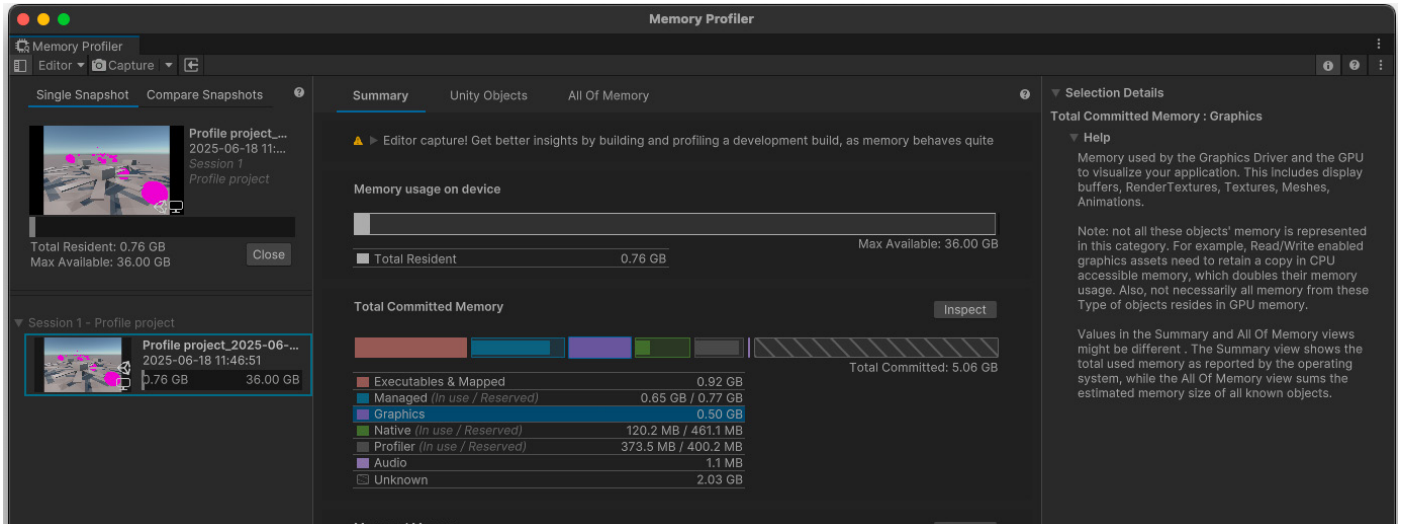
The Summary tab gives you a high-level snapshot of your project's memory usage at the moment a memory capture was taken. It's perfect for when you want a fast and informative overview without diving into detailed analysis.

This view highlights key metrics and can help you quickly spot potential memory issues or unexpected usage patterns. It's especially useful when comparing snapshots or debugging memory usage over time. Let's look at a few of its key sections.



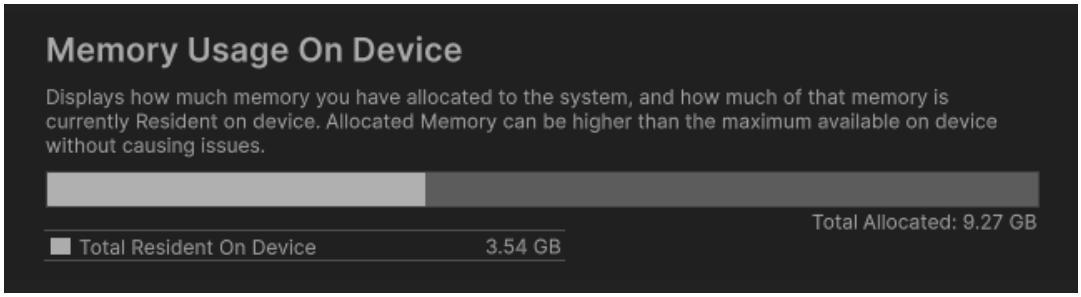
The Summary tab displays an overview of memory at the time the snapshot was captured.

Tips: In the right panel (see image below), you'll find helpful contextual information about your snapshot. These can alert you to possible problems or guide you in interpreting results.

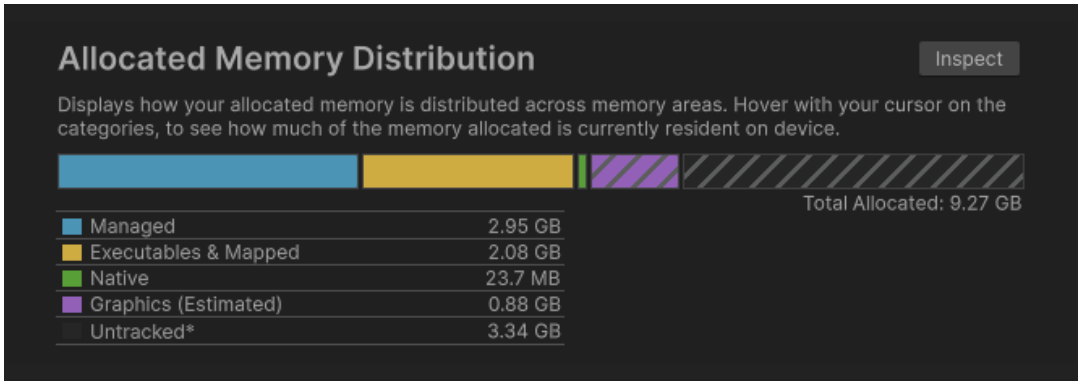


The right panel provides helpful hints about your snapshot.

Memory Usage on Device: This shows the application footprint in physical memory. It includes all Unity and non-Unity allocations resident in memory at the time of the capture.

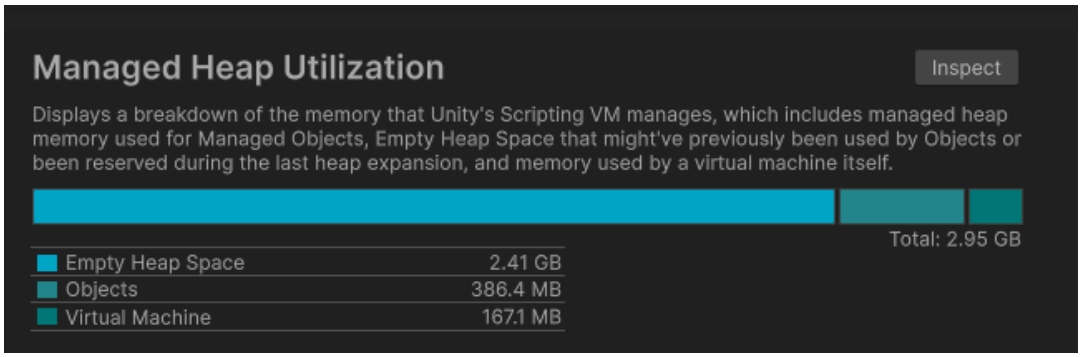


Allocated Memory Distribution: This view visualizes how allocated memory is distributed across different memory categories.

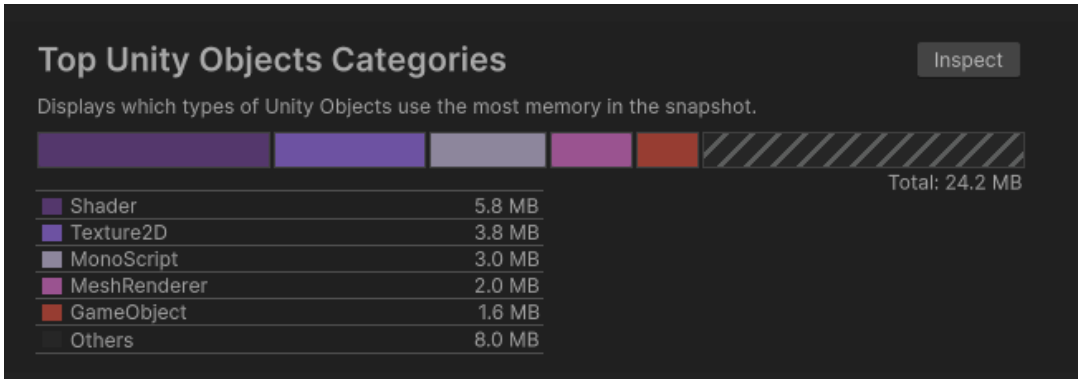


Note the Untracked* memory bar. It corresponds to the memory that Unity does not track through its memory management system. Such allocations may come from native plugins and drivers. Use the platform-specific profiler to analyze Untracked memory usage for your target device.

Managed Heap Utilization: In this view, you'll get a breakdown of the memory that Unity's scripting VM manages, which includes managed heap memory used for managed objects, empty heap space that might have previously been used by objects or been reserved during the last heap expansion, and memory used by a virtual machine itself.

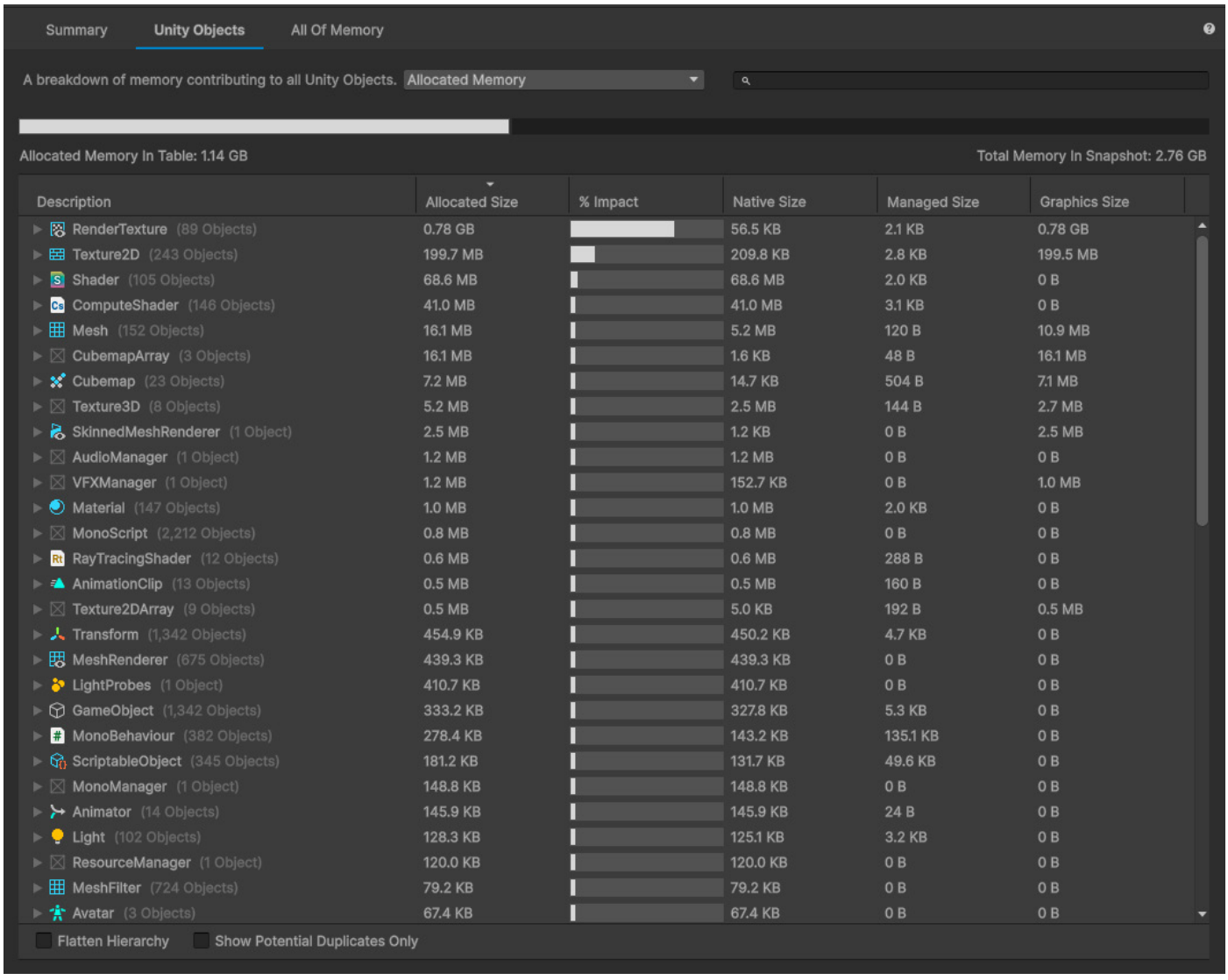


Top Unity Object Categories: This displays which types of Unity objects use the most memory in the snapshot (e.g., Texture2D, mesh, GameObject).



Unity Objects tab

The Unity Objects tab displays any Unity objects that allocated memory, how much native and managed memory the object uses, and the combined total. Use this information to identify areas where you can eliminate duplicate memory entries or to find which objects use the most memory. And via the search bar, you can find the entries in the table which contain the text you enter.



The Unity Objects tab allows you to drill down into captured snapshot memory usage with high granularity.

By default, the table lists all relevant objects by Allocated Size in descending order. You can click on a column header name to sort the table by that column or to change whether the column sorts in ascending or descending order.

Use this to your advantage when optimizing memory usage and aiming to pack memory more efficiently for hardware platforms where memory budgets are limited.

Memory profiling techniques and workflows

Begin by analyzing a Memory Profiler snapshot to identify high-memory usage areas. Once you capture or load a Memory Profiler snapshot, use the Unity Objects tab to inspect the categories, ordered from largest to smallest in memory footprint size.

Project assets are often the highest consumers of memory. Using the Table mode, locate textures, meshes, audio clips, render textures, shader variants, and preallocated buffers. These are often good candidates to start with when optimizing memory usage. The Project Auditor is a great complementary tool here as it can provide some recommendations about how to reduce memory use for assets (making sure the asset is set up properly in the **Import Settings** Inspector is a good starting place).

Locating memory leaks

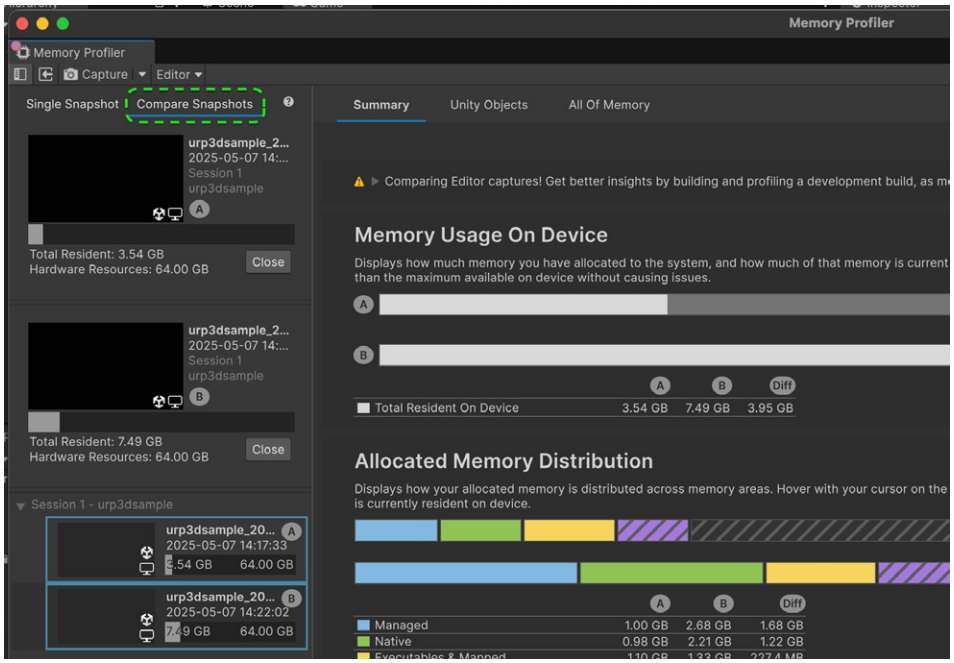
A memory leak is a situation where unused assets, objects, or resources are not properly released from memory. This can lead to progressively increasing memory usage and performance issues or crashes.

A memory leak typically happens when:

— An object is not released manually from memory through the code.

— An object unintentionally remains in memory because another object still holds a reference to it.

The Memory Profiler has a Compare Snapshots mode which can help find memory leaks by comparing two snapshots over a specific time frame. This comparison can reveal objects that persist in memory when they should be deallocated.



Compare two snapshots to see the diffences.

A frequent scenario for memory leaks in Unity games is after unloading a scene. Objects from the unloaded scene might not be correctly garbage collected if references to them still exist.

Locating recurring memory allocations over application lifetime

Through differential comparison of multiple memory snapshots, you can identify the source of continuous memory allocations during application lifetime.

Here are some tips to help identify managed heap allocations in your projects.

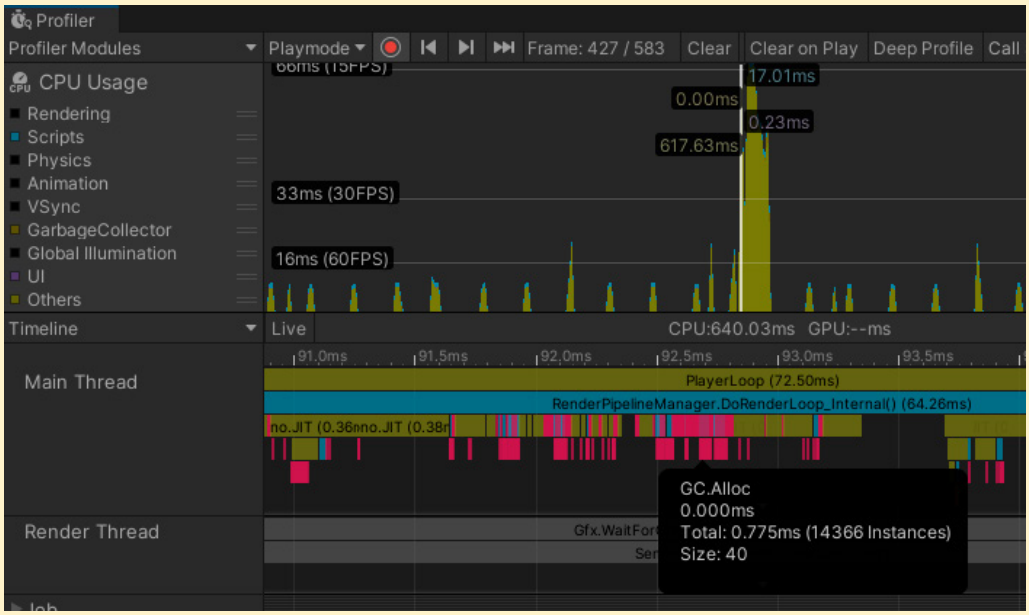
Memory Profiler module in the Unity Profiler

The Memory Profiler module in the Unity Profiler represents managed allocations per frame with a red line. This should be 0 most of the time, so any spikes in that line indicate frames you should investigate for managed allocations.

Any spikes seen for GC Allocated In Frame give you pointers to investigate for managed allocations.

Timeline view in the CPU Usage Profiler

The Timeline view in the CPU Usage Profiler shows allocations, including managed ones, in pink, making them easy to see and hone in on.



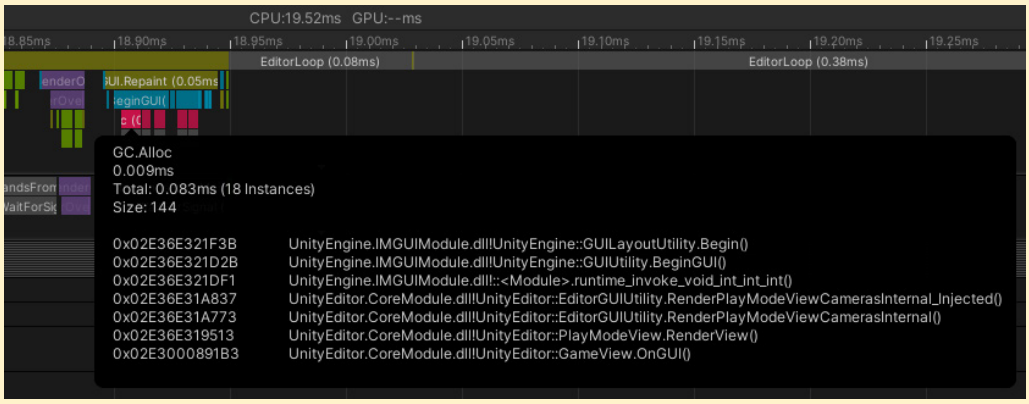
Managed allocations appear as pink-colored markers in the Timeline view.

Allocation Call Stacks

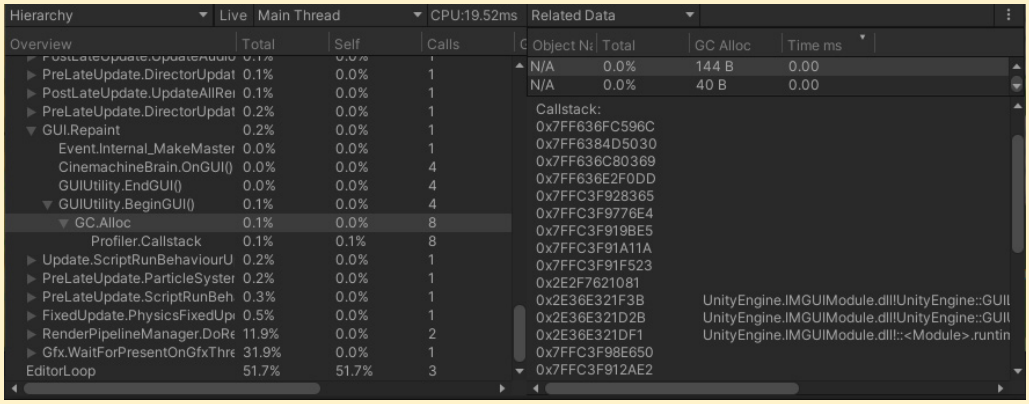
Allocation call stacks provide a quick way to discover managed memory allocations in your code. These will provide the call stack detail you need at less overhead compared to what deep profiling would normally add, and they can be enabled on the fly using the standard Profiler.

Allocation call stacks are disabled by default in the Profiler. To enable them, click the **Call Stacks** button in the main toolbar of the Profiler window. Change the **Details** view to **Related Data**.

Note: If you're using an older version of Unity (prior to allocation call stack support), then deep profiling is a good way to get full call stacks to help find managed allocations.



Enable allocation call stacks in the Profiler to follow the call stack back to the source for managed allocations.

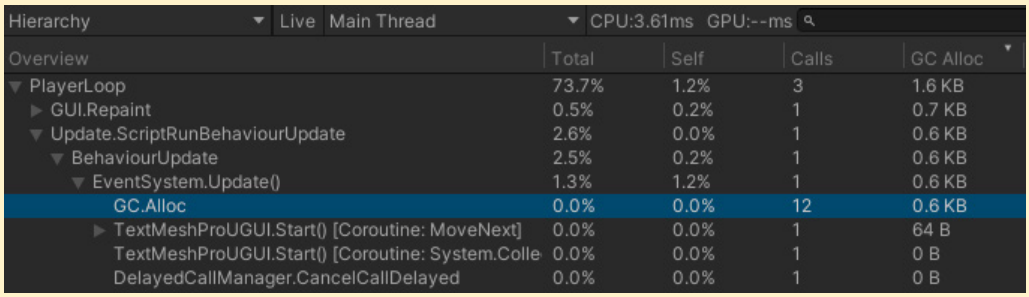


The Related Data panel in the Hierarchy view also reveals allocation call stack details.

GC.Alloc samples selected in the Hierarchy or Raw Hierarchy will now contain their call stacks. You can also see the call stacks of GC.Alloc samples in the selection tooltip in Timeline.

The Hierarchy view in the CPU Usage Profiler

The Hierarchy view in the CPU Usage Profiler lets you click on column headers to use them as the sorting criteria. Sorting by GC Alloc is a great way to focus on those.



Use the Hierarchy view in the CPU Usage Profiler module to filter and focus on managed allocations.

Memory and GC optimizations

Reduce the impact of garbage collection (GC)

Unity uses the Boehm-Demers-Weiser garbage collector to automatically clean up memory when it's no longer needed for your application. The GC stops running your program code and only resumes normal execution once its work is complete.

While the automatic management is convenient, unnecessary or frequent allocations can lead to performance hiccups because the garbage collector has to pause your game to clean up unused memory (also known as GC spikes). Here are some common pitfalls to keep in mind:

— **Strings:** In C#, strings are reference types, not value types. This means that every new string will be allocated on the managed heap, even if it's only used temporarily. Reduce unnecessary string creation or manipulation. Avoid parsing string-based data files such as JSON and XML, and store data in ScriptableObjects or formats like MessagePack or Protobuf instead. Use the StringBuilder class if you need to build strings at runtime.

— **Unity function calls:** Some Unity API functions create heap allocations, particularly ones which return an array of temporary managed objects. Cache references to arrays rather than allocating them in the middle of a loop. Also, take advantage of certain functions that avoid generating garbage. For example, use **GameObject.CompareTag** instead of manually comparing a string with **GameObject.tag** (as returning a new string creates garbage).

  You can also use the Project Auditor to list these alternatives; this can help ensure that you're using the non-allocating versions wherever possible.

— **Boxing:** Boxing occurs when a value type (e.g., int, float, struct) is converted to a reference type (e.g., object). Avoid passing a value-typed variable in place of a reference-typed variable. This creates a temporary object, and the potential

garbage that comes with it implicitly converts the value type to a type object (e.g., **int i = 123; object o = i**). Instead, try to provide concrete overrides with the value type you want to pass in. Generics can also be used for these overrides.

- **Coroutines:** Though yield does not produce garbage, creating a new WaitForSeconds object does. Cache and reuse the WaitForSeconds object rather than creating it in the yield line.
- **LINQ and Regular Expressions:** Both of these generate garbage from behind-the-scenes boxing. Avoid LINQ and Regular Expressions if performance is an issue. Write for loops and use lists as an alternative to creating new arrays.
- **Generic Collections and other managed types:** Don't declare and populate a List or collection every frame in Update (for example, a list of enemies within a certain radius of the player). Instead, make the List a member of the MonoBehaviour and initialize it in Start. Simply empty the collection with **Clear** every frame before using it.

Time garbage collection whenever possible

If you are certain that a garbage collection freeze won't affect a specific point in your game, you can trigger garbage collection with System.GC.Collect. A classic example of this is when the user is in a menu or pauses the game, where it won't be noticable.

See Understanding Automatic Memory Management for examples of how to use this to your advantage.

Use the Incremental Garbage Collector to split the GC workload

Rather than creating a single, long interruption during your program's execution, incremental garbage collection uses multiple, shorter interruptions that distribute the workload over many frames. If garbage collection is causing an irregular frame rate, try this option to see if it reduces the problem of GC spikes. Use the Profile Analyzer to verify its benefit to your application.

Note that using the GC in Incremental mode adds read-write barriers to some C# calls, which comes with some overhead that can add up to ~1 ms per frame of scripting call overhead. For optimal performance, it's ideal to have no GC Allocs in the main gameplay loops so that you don't need the Incremental GC for a smooth frame rate and can hide the GC.Collect where a user won't notice it, for example, when opening the menu or loading a new level. In such optimized scenarios, you can perform full, non-incremental garbage collections (using System.GC.Collect()).

To learn more about the Memory Profiler check out the following resources:

- Memory Profiler Walkthrough and Tutorial
- Memory Profiler documentation
- Improve memory usage with the Memory Profiler in Unity
- Memory Profiler: The Tool for Troubleshooting Memory-related Issues
- Working with the Memory Profiler

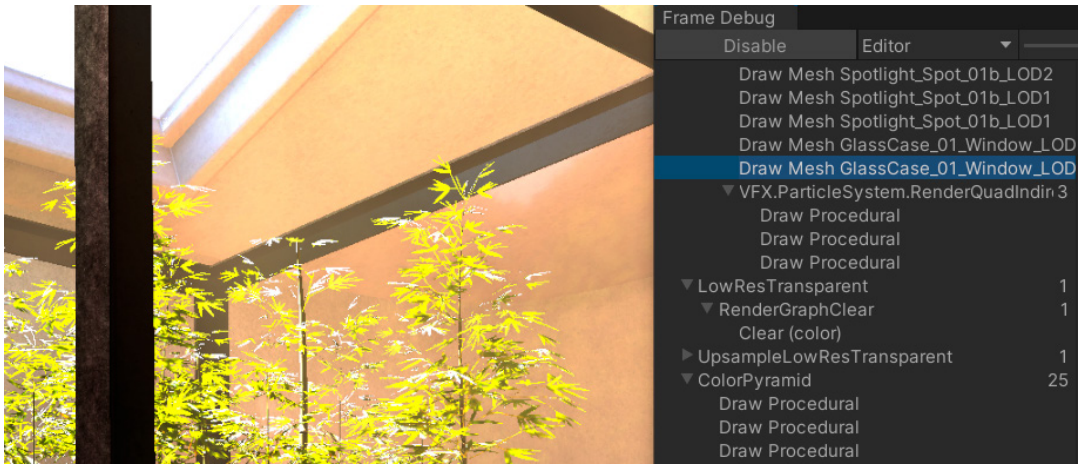
Frame Debugger

The Frame Debugger helps you debug and optimize rendering by letting you freeze playback for a running game on a specific frame and view the individual draw calls used to render it. You can step through the list of draw calls, one by one, to see the frames as they are constructed to form a scene from its graphical elements.

The Frame Debugger makes it easy to see where a draw call corresponds to the geometry of a GameObject. It highlights the GameObject in the main Hierarchy panel to assist with identification.

Understanding draw calls:
A draw call in Unity is a request sent from the CPU to the GPU to render a specific set of geometry (like a mesh, skybox, user interface, etc.) with a particular material and shader. Each time Unity needs to render a different object, material, or state change, it issues a new draw call.

The Frame Debugger can also be used to test for overdraw by analyzing the rendering order frame-by-frame. See the optimization tips below for more information.

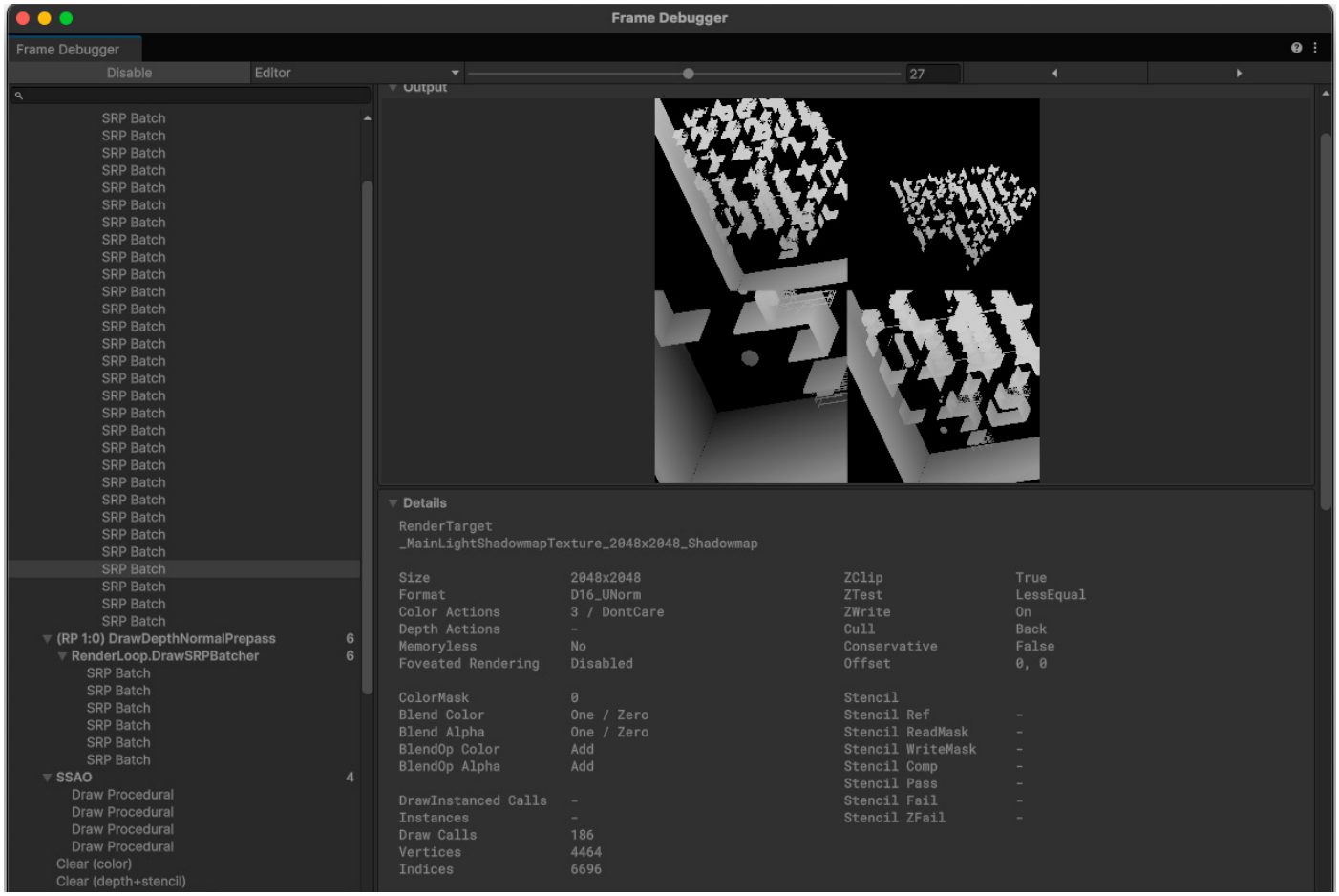


Use the Frame Debugger to analyze how identified overdraw occurs.

Open the Frame Debugger from the **Window > Analysis > Frame Debugger** menu.

With your application running in the Editor or on a device, click **Enable**. This will pause the application, and all the draw calls for the current frame will be captured and listed in sequence on the left side of the Frame Debugger window. The capture will include additional details, such as framebuffer clear events.

The slider at the top of the Debugger window lets you scrub rapidly through the draw calls to quickly locate an item of interest.



The Frame Debugger window lists draw calls and events on the left side and provides a slider to visually step through each one.

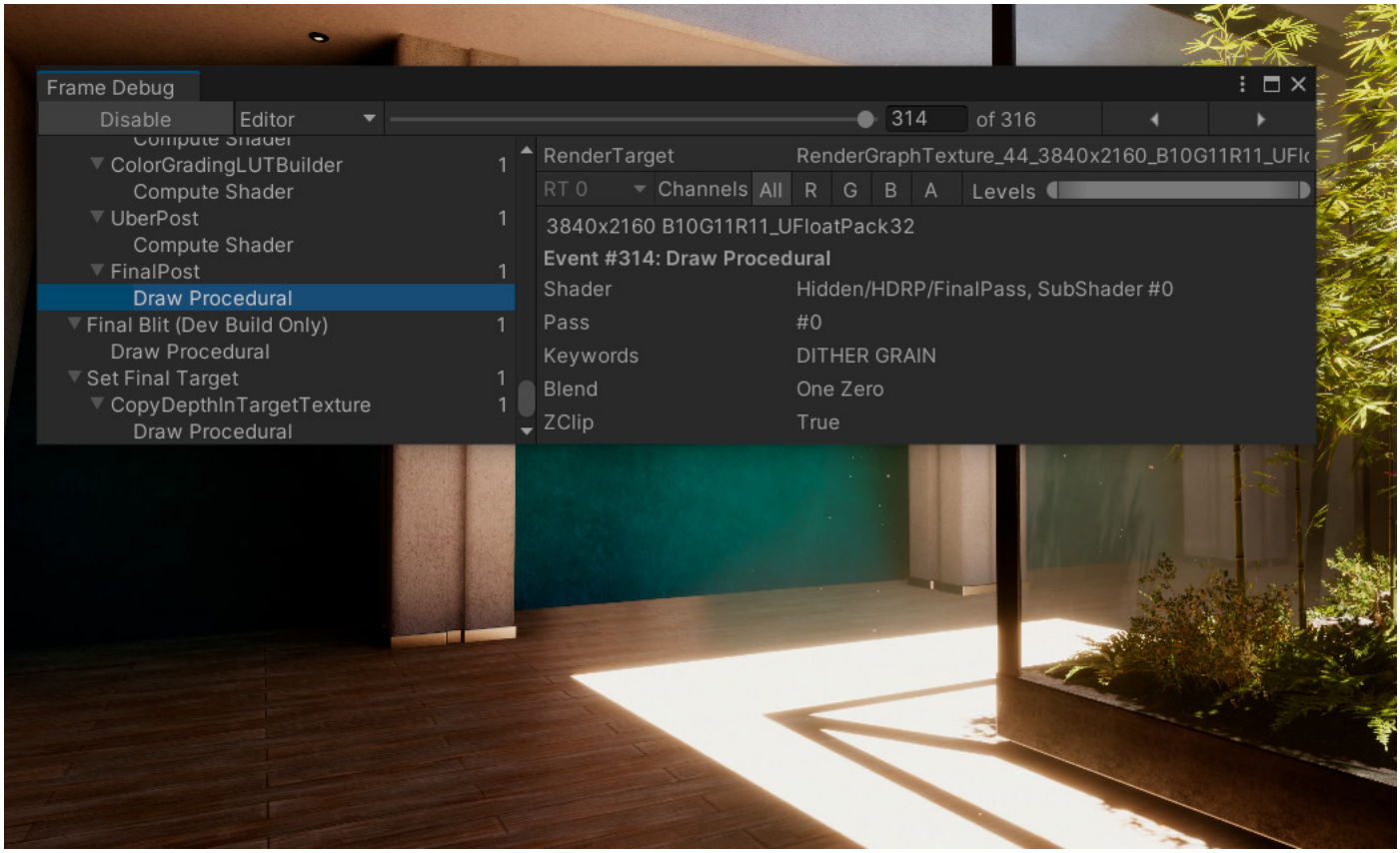
Unity issues draw calls from the CPU to the graphics API to draw geometry on the screen. A draw call tells the graphics API what to draw and how. Each draw call contains all the information the graphics API needs, such as information about textures, shaders, and buffers. Often, the preparation for a draw call is more resource-intensive than the draw call itself.

This preparation process is known as the render state, and you can improve its performance by minimizing changes to it.

Use the Frame Debugger to identify where draw calls originate and visualize the rendering process. This helps inform your decisions about how to group draw calls to reduce render state changes.

Reference the Frame Debugger's list hierarchy to locate where interesting draw calls originate from. Selecting an item from the list will show the scene (in the Game window) as it appears up to and including that draw call.

Minimizing draw calls is crucial for performance, especially on mobile or VR platforms, because each call adds CPU overhead. Techniques like batching, GPU instancing, and texture atlasing help reduce the number of draw calls by combining objects that share materials or meshes.



The Game window displays a scene frame constructed up to and including the selected draw call (near the end of applying post-processing effects) in the Frame Debugger. The panel to the right of the list hierarchy provides information about each draw call, such as the geometry details and the shader used for rendering.

Other useful information includes reasons for why a draw call couldn't be batched with previous ones, and a breakdown of the exact property values that were fed into shaders.

Remote Frame Debugging

You can remotely debug frames by attaching the Frame Debugger to a running Unity Player on supported platforms (WebGL is not supported). For Desktop platforms, enable **Run In Background** for builds.

To set up remote frame debugging:

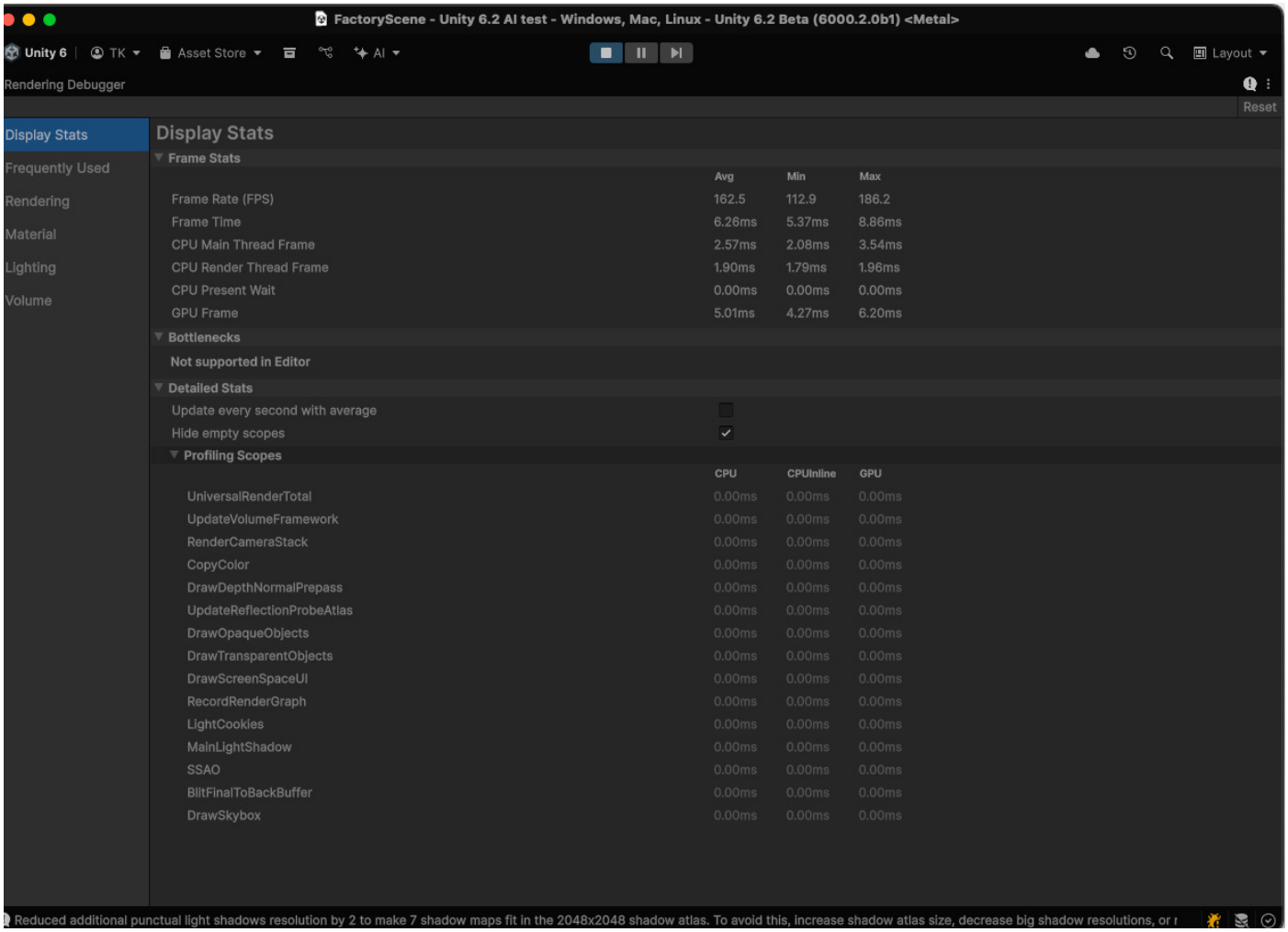
1. Create a standard build of the project to your target platform (select Development Build).
2. Run the player.
3. Open the Frame Debugger window from the Editor.
4. Click the Player selection dropdown and choose the active player that is running.
5. Click the Enable button.

You can now step through draw calls and events in the Frame Debug list hierarchy and observe the results in the active player.

Rendering Debugger

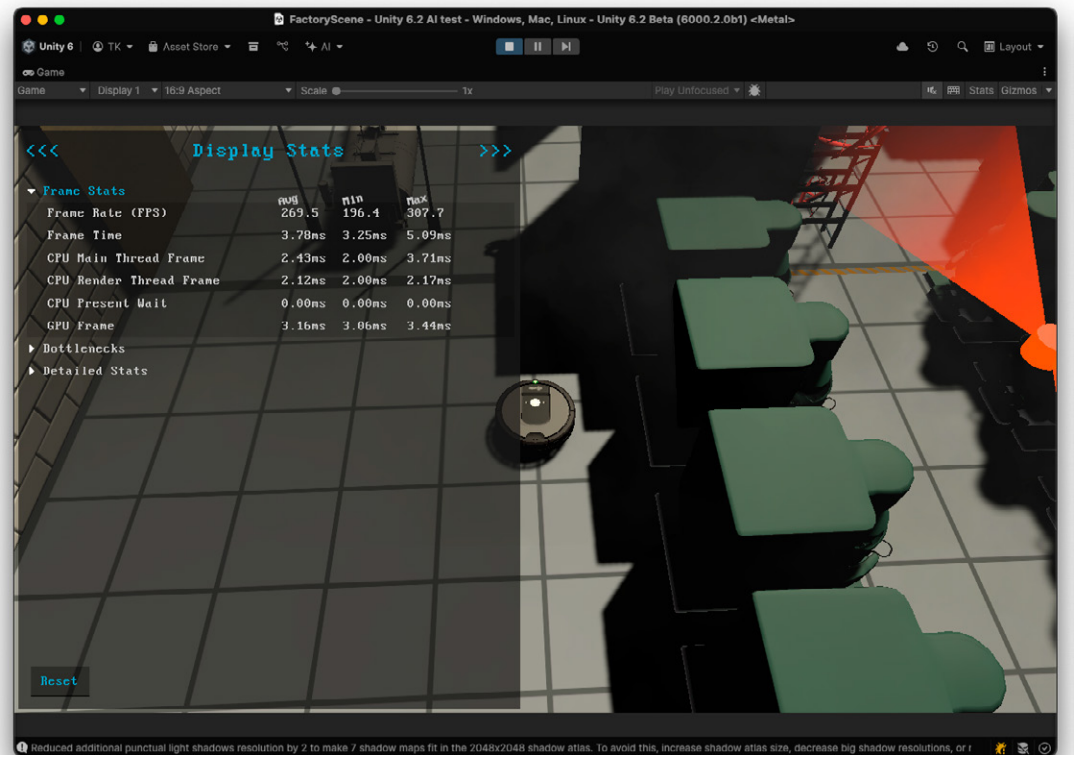
The Rendering Debugger provides multiple debug views and modes that display information about overdraw, lighting complexity, rendering, and material properties, allowing you to pinpoint rendering issues and optimize scenes for URP and HDRP.

To open the tool go to **Window > Analysis > Rendering Debugger** in the Editor. You can also use the shortcut **LeftCtrl+Backspace** (**LeftCtrl+Delete on macOS**) in playmode or for desktop player build.



The Rendering Debugger lets you visualize various lighting, rendering, and material properties so you can identify rendering issues and optimize scenes.

This can help in diagnosing visual artifacts or performance bottlenecks related to rendering. Note that the window with the detailed statistics is only available for Development builds, and there can be limitations with how it interacts with non-pipeline-specific shaders or external rendering objects.



You can open the Rendering Debugger window in the Editor, or as an overlay in Game view, Play mode, or your built application.

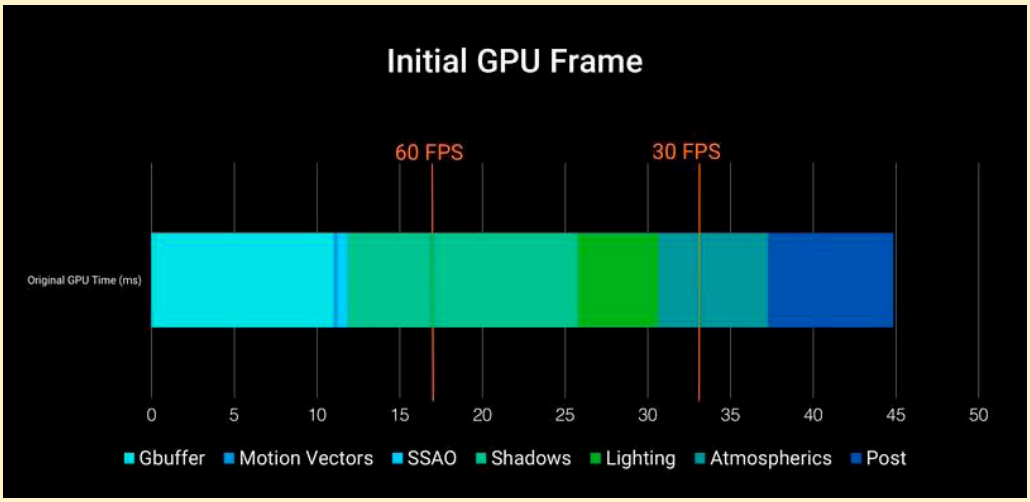
Five rendering optimizations for common pitfalls

Use these tips and tricks to optimize common rendering performance issues that can be identified using the Frame Debugger and other render debug tools.

Identify your performance bottlenecks first

To begin, locate a frame with a high GPU load. The majority of platforms provide solid tools for analyzing your project's performance on both the CPU and the GPU. Examples include Arm Performance Studio for Arm hardware / Immortalis and Mali GPUs, PIX for Microsoft Xbox, Razor for Sony PlayStation, and Xcode Instruments for Apple iOS.

Use your respective native profiler to break down the frame cost into its specific parts. This is your starting point to improve graphics performance.



This view was GPU-bound on a PS4 Pro at roughly 45 ms per frame.

Draw call optimization

PC and current generation console hardware can push a lot of draw calls, but the overhead of each call is still high enough to warrant trying to reduce them. On mobile devices, draw call optimization is often vital. Draw call batching is a method that combines meshes so that Unity can render them in fewer draw calls.

Use the Frame Debugger as explained above to help identify draw calls that can be reorganized for optimal group and batch. The tool also helps to identify why certain draw calls can't be batched.

Techniques to help reduce draw call batches include:

— **Occlusion Culling** removes objects hidden behind foreground objects and reduces overdraw (when the GPU redraws the same pixel multiple times due to overlapping transparent objects) of the non-visible elements. Be aware this requires additional CPU processing, so use the Profiler to ensure moving work from the GPU to CPU is beneficial and that you are not creating new bottlenecks.

— **GPU instancing** reduces drawcalls by rendering many objects that share the same mesh and material in fewer batches. It allows complex scenes with fewer performance costs and minimal visual repetition.

— The **SRP Batcher** reduces the GPU setup between draw calls by batching Bind and Draw GPU commands. To benefit from SRP batching, use as many Materials as needed, but restrict them to a small number of compatible shader variants, e.g., Lit and Unlit Shaders in URP and HDRP, with as few variations between keyword combinations as possible.

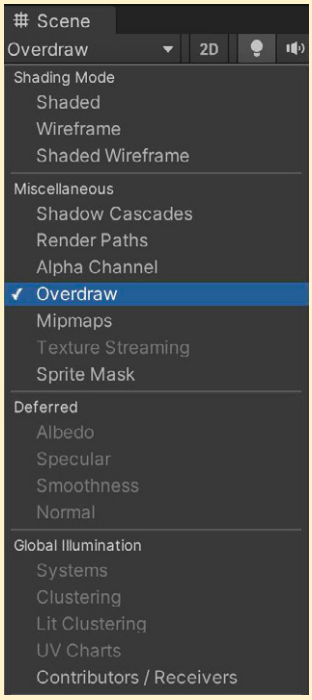
— **GPU Resident Drawer** uses GPU instancing to draw many GameObjects, which significantly reduces the number of draw calls. This frees up CPU processing time by shifting more of the rendering workload to the GPU, leading to improved performance, especially in scenes with many similar objects.

Optimize fill rate by reducing overdraw

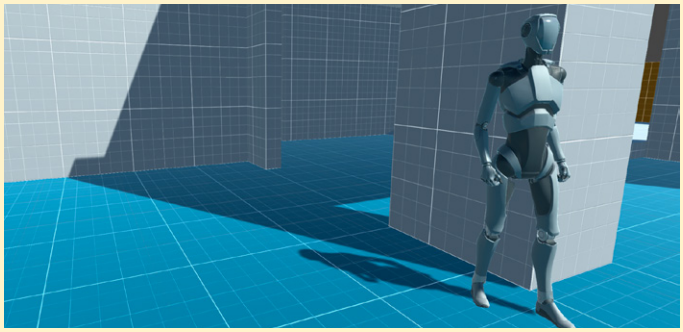
Objects rendering on top of one another create overdraw. Overdraw can indicate an application is trying to draw more pixels per frame than the GPU can cope with. Not only is performance at risk, but thermals and battery life on mobile devices suffer too. You can combat overdraw by understanding how Unity sorts objects before rendering them.

The Built-In Render Pipeline sorts GameObjects according to their Rendering Mode and renderQueue. Each object's shader places it in a render queue, which often determines its draw order.

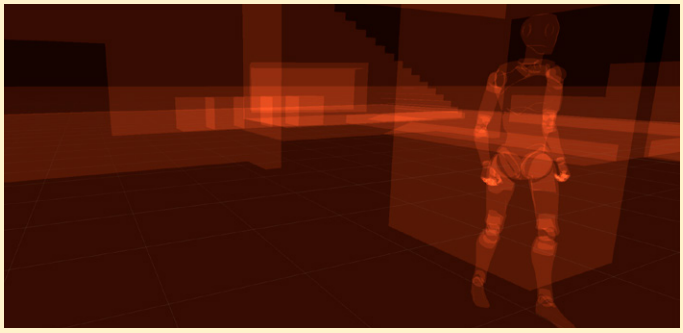
If you're using the Built-In Render Pipeline, use the Scene view control bar to visualize overdraw. Switch the draw mode to Overdraw.



Overdraw in the Scene view control bar



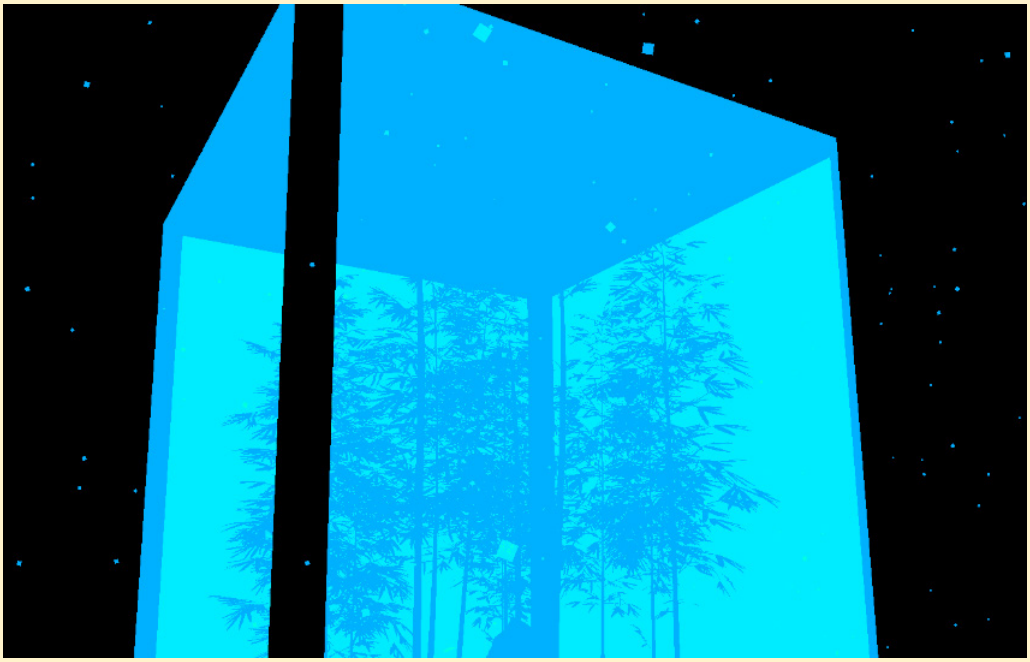
A scene in Standard shaded view



The same scene as above, now in Overdraw view; overlapping geometry is often a source of overdraw

HDRP controls the render queue slightly differently. Read the section on Renderer and Material Priority to understand this approach in greater detail.

You can use the Rendering Debugger for identifying overdraw in URP and HDRP as we described above.



Visualizing overdraw with HDRP and the Fullscreen Debug Mode

Examine your most expensive shaders

This is a deep topic, but in general, aim to reduce shader complexity where possible. Some easy wins here involve reducing precision where possible, i.e., use half precision floating point variables if you can. You can also learn about wavefront occupancy for your target platform and how to use GPU profiling tools to assist in getting a healthy occupancy.

Multi-core optimization for rendering

Enable Graphics Jobs in **Player Settings > Other Settings** to take advantage of the multi-core processors on desktop and consoles. Graphics Jobs allows Unity to spread the rendering work across multiple CPU cores, removing pressure from the render thread. See this Multithreaded Rendering and Graphics Jobs tutorial for details.

Profile post-processing effects

Ensure that your post-processing assets are optimized for your target platform. Tools from the Unity Asset Store that were originally authored for PC games might consume more resources than necessary on consoles or mobile devices. Profile your target platform using its native profiler tools. When authoring your own post-processing effects for mobile or console targets, keep them as simple as possible.

There are many more tools available to help with frame debugging and analysis. Take a look at the profiling and debug tools index for further inspiration.

To learn more about about the Unity Frame Debugger, check out the following resources:

— Unity Frame Debugger documentation

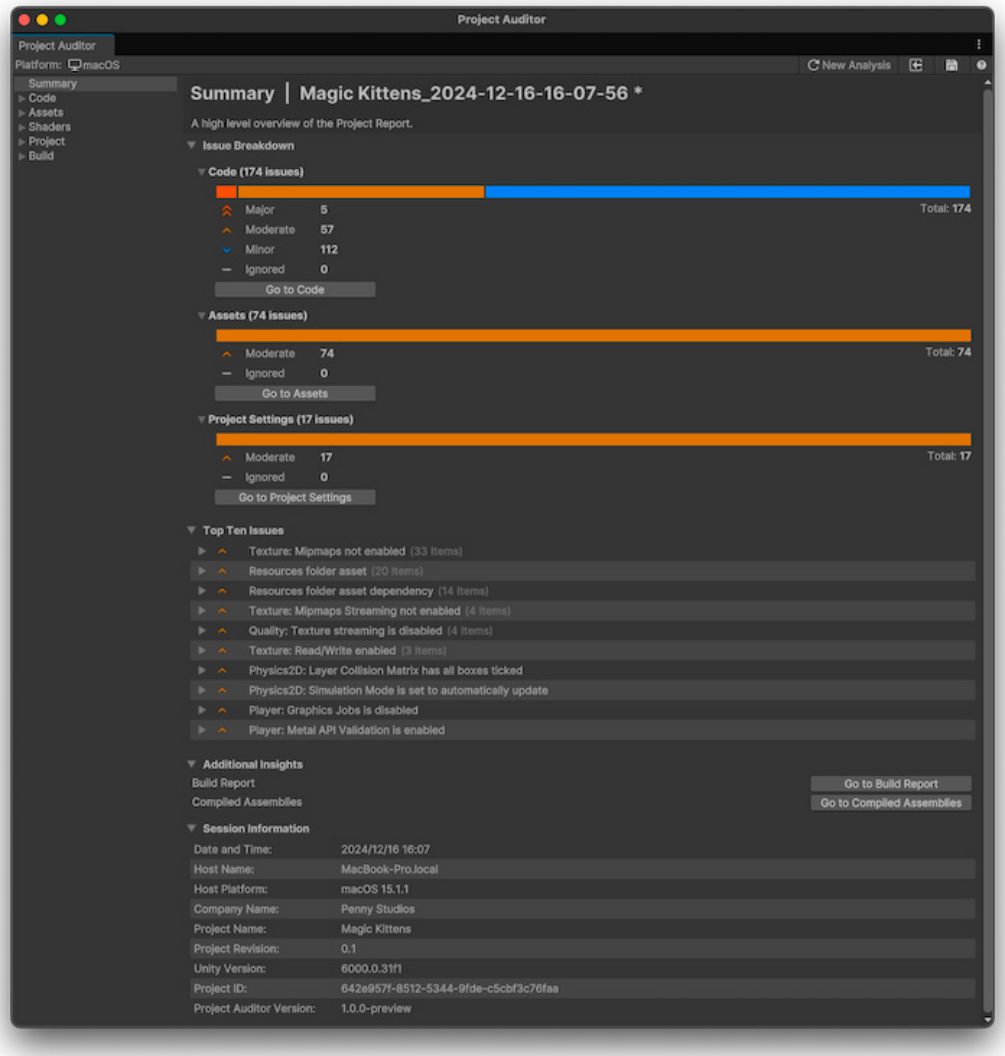
— Working with the Frame Debugger

— Profiling Rendering

Project Auditor

The Project Auditor, introduced as a package in Unity 6.1, is a powerful analysis tool for Unity projects, designed to help developers optimize performance, maintain best practices, and identify potential issues and bottlenecks in their projects.

Project Auditor scans your entire project and provides detailed reports about inefficiencies, such as heavy scripting calls, unused assets, excessive entity counts, etc.



The Project Auditor Summary view

The Project Auditor covers several different areas:

— **Performance optimization:** It identifies problems that could impact your project's runtime performance, such as excessive garbage generation, unnecessary object allocations, or expensive function calls.

— **Code and asset review**: It highlights unused assets, inefficient code patterns, or outdated APIs that can be refactored. This helps reduce build size, improve overall project maintainability, and optimize memory use.

— **Diagnostics and best practices**: It provides recommendations based on Unity best practices and highlights errors or warnings related to your project setup, like missing references, or suboptimal Player or Quality settings.

— **Customizable reports**: It organizes the results into categories, making it easy to prioritize optimizations. You can also create custom rules to tailor the analysis to your specific project or needs.

💡**Tips**:

— Run the Project Auditor at key stages of development (e.g., before milestones, beta releases, final builds). Regular audits help catch performance bottlenecks, unused assets, or outdated code early, preventing problems from growing larger as your project scales.

— You can automate running Project Auditor as part of your CI or build setup (as shown here in the manual) and use the reports to make sure no one checks in any assets or code that add new issues (using the API detailed here).

— You can add your own rules if there are particular things you want to make sure you catch in your game; e.g. texture settings, sizes, or more complicated rules. See this page in the manual for more details about how to do this.

The reports generated by the Project Auditor are categorized by severity (Major, Moderate, and Info). Focus on the most severe issues first, as they often highlight performance-critical problems, such as over-allocation of memory or excessive garbage collection. They're also likely to be in code paths that are called more frequently, like Update, where any performance problems they bring will be more obvious to players.

The Project Auditor also checks settings like **Player settings** and **Quality settings** and makes recommendations about what you might change. Use this to ensure your build targets, resolution, text compression, or other project settings are optimized for your intended platform.

Domain Reload

The Unity Editor allows you to configure settings about entering Play mode; this page has more details about it, but you can often speed up your Editor iteration time by disabling Domain Reload. However, this will no longer reset your scripting state every time you enter Play mode, so you have to do this manually in your code.

The Code area in Project Auditor can analyze the scripts in your project to help you find anywhere that you need to reset your script variables. It's considered best practice to fix all the issues displayed in the Domain Reload view and then to disable domain reload. To populate this view with data, you must enable the Use Roslyn Analyzers setting in the Preferences window. Then you can run through the list of issues, following the instructions in the manual to fix them. Once they're all addressed you can disable Domain Reload when entering play mode.

Deep profiling

As mentioned in the Profiling 101 section, by default Unity only profiles code that's explicitly wrapped in Profiler markers. This includes the first call stack depth of managed code invoked by the engine's native code.

Enabling Deep Profiling will result in the insertion of Profiler markers at the beginning and end of each function call. This allows a great deal of detail to be captured. Use the Deep Profile setting to work out what's happening inside long Profiler markers that don't show enough of their call stacks.

This granular approach to measuring game performance can be preferable to snapshot-based profiling (sample profiling), which has the potential to miss detail in captures.

Be sure to check out the ProfileMarker API as a way to manually instrument problematic areas of code.

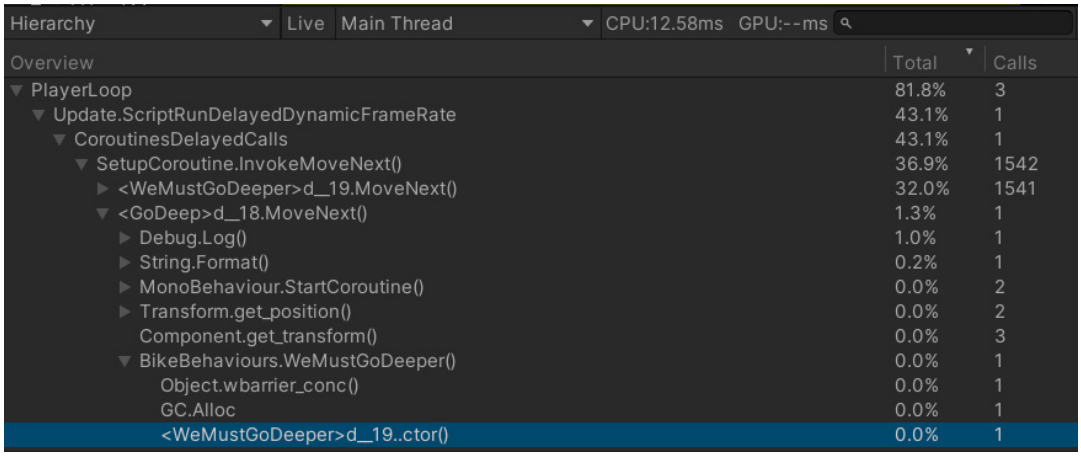
They can have a much lower performance impact than deep profiling. Sometimes it's even quicker to add a ProfileMarker and rebuild your game than it is to get to the part of the game you want to test with Deep Profiling enabled.

Another alternative to get full call stacks on a device build is to run a native CPU Usage Profiler. In some cases, this is easier and less intrusive to performance than deep profiling.

When to use deep profiling

You should only enable the Deep Profile setting once you have identified the specific part of your application or managed code that needs to be examined in greater detail. Deep profiling is resource-intensive and consumes a lot of memory. Your application will run slower when it's enabled.

Deep profiling allows you to traverse down the call tree in detail and spot inefficiencies or problems in your code.



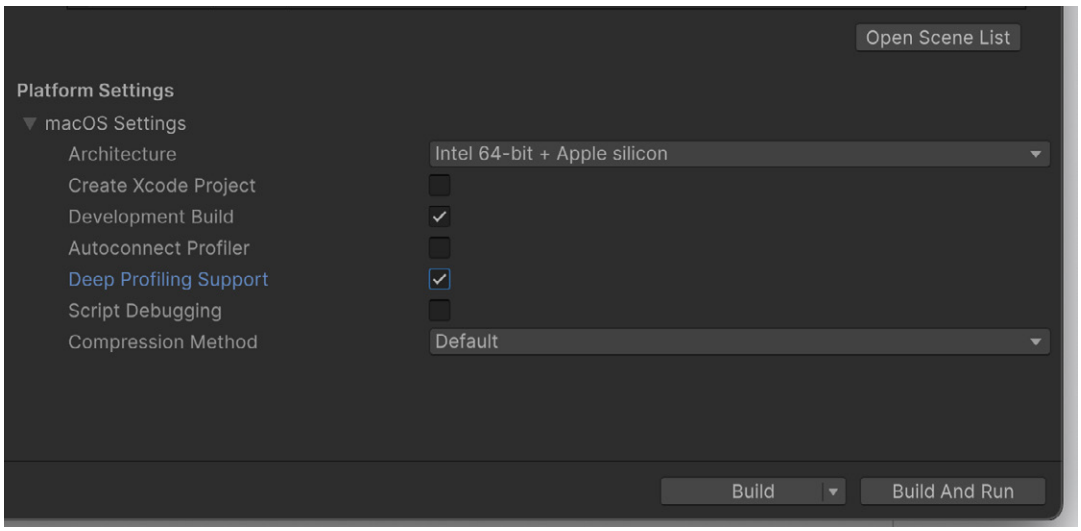
Deep profiling opens up a huge amount of detail when you need to trace and understand specific issues.

However, deep profiling adds a marker to the start and end of every function call, and each marker adds some overhead. This means that a part of your code which has a deep callstack (say MyDeepFunction) will show up as more expensive than places which do all their work inside a single function (MySingleFunction). That means you cannot rely on the relative timing of these two pieces of code – MyDeepFunction might look more expensive than MySingleFunction with deep profiling enabled, but this cost could all be in the extra markers added.

Note: Support for deep profiling in both the Mono and IL2CPP backends was added from Unity 2019.3 onward, which is great news for platforms where IL2CPP is mandatory, such as iOS.

Using deep profiling

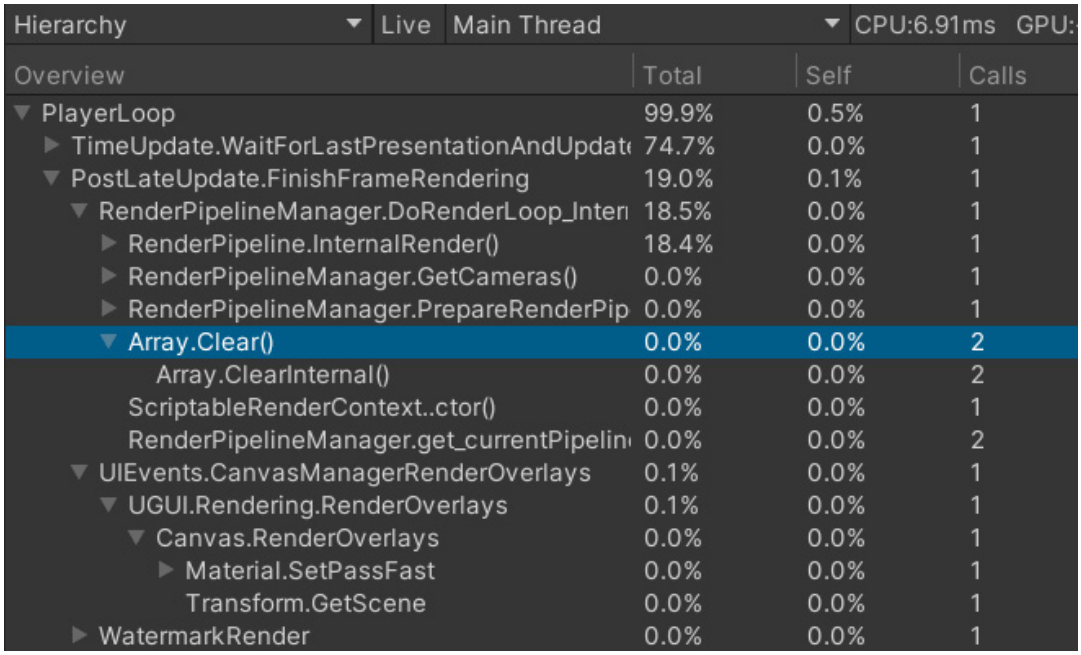
To use deep profiling with player builds, you'll need to enable it via **File > Build Settings > Enable Deep Profiling Support.**



Enabling Deep Profiling Support

Once support is enabled, you can easily toggle Deep Profiling on or off for your build in the Profiler window as needed.

A Deep Profile button that's faded out when attached to the player indicates that Deep Profiling Support was not enabled for your build.



Deep profiling reveals much more information about the performance and timing of your application code. It shows the full method call tree, helping you dig into where managed allocations are happening.

Deep profiling tips

Top-to-bottom approach

When profiling your application, start at a high level and try to locate areas where performance can be improved without using deep profiling. As you need more information, you can enable Deep Profiling in the Profiler to dig in at a more granular level. Using this approach will help to keep the level of information being displayed in the Profiler Hierarchy to a minimum, allowing you to focus on the goal at hand.

Deep profile only when necessary

In general, it's best to use deep profiling when you need to get much lower-level detail about the performance of your code. While leaving the Deep Profiling flag enabled for builds will not affect performance without actually toggling the feature to enabled, when it is enabled, it causes your application to run slowly.

If you are only interested in finding the source of managed allocations in your code, remember that versions of Unity from 2019.3 and newer allow you to do this without the need to enable Deep Profiling. Use the Call Stacks toggle and Calls dropdown in the Profiler to help locate managed allocations.

Deep profiling in automated processes

To toggle Deep Profiling on when profiling from the command line, add the ***-deepprofiling*** argument to your build executable. For Android / Mono scripting backend builds use the adb command line argument like this:

adb shell am start -n com.company.game/com.unity3d.player.UnityPlayerActivity -e 'unity' '-deepprofiling'

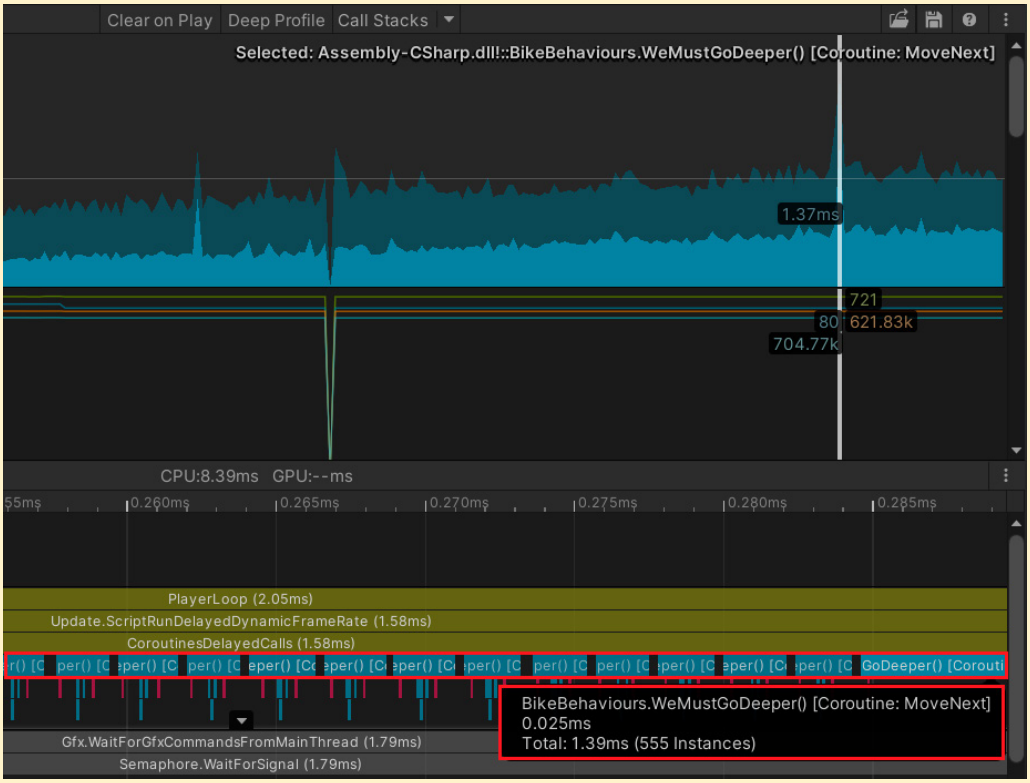
Deep profiling on low-spec hardware

Lower-spec hardware has limited memory and performance that can affect your ability to use deep profiling. Unity's Profiler samples are stored in a ring buffer, which can fill up when using the Deep Profile setting on slower devices. If this happens, Unity will display an error message.

You can allocate more memory to the Profiler for this buffering data by setting the **Profiler.maxUsedMemory** property (bytes). The default is 128 MB for Players and 512 MB for the Editor. Increase this as required on slower-device Player builds if you run into problems when deep profiling.

If you need to profile code in higher detail on hardware that runs too slowly (or not at all) due to the overhead that deep profiling adds, you can profile deeper with your own markers.

Instead of enabling the Deep Profile setting, add Profiler markers to the specific areas of interest in your code. These markers will appear in the Profiler Timeline or Hierarchy when viewing the CPU Usage module.



Add Profiler markers to profile deeper layers of code when deep profiling adds too much overhead.

Which profiling tools to use and when?

Profiling provides the best benefit when started at the beginning of a project lifecycle. By starting early, you can establish baselines that are useful for comparisons at checkpoints further into your game and application development. It's important to know which tool to select from the "profiling tool belt" and when.

Once you understand the uses and benefits of each tool, it will be easier for you to know when to use them. Be sure to learn about each profiling tool Unity has to offer in the Unity profiling and debug tools section.

To help answer the "when," here is a list of checkpoint ideas in a project's life cycle, which may be useful to reference when planning a profiling strategy.

— **Prototyping**: Profiling is important to reduce risk in the prototype stage of a project. If the game design document calls for 10,000 enemies onscreen, you need to be able to build and profile a prototype that proves such a thing is possible on the target platform. If it's not, you need to change the design.

— **Early stages of the project**: Establish a baseline for project performance across a selection of target device hardware. Get a rough idea of memory usage using the

Memory Profiler, and ensure that plans for the project's scope are not trending to a point where memory limits on target hardware will become an issue further on.

- **End of sprint**: If you're working in sprints on an Agile team, then the end-of-sprint release candidate (RC) is a great point at which to run a standardized suite of profiling tools. Ensure you have a standard format to record results and metrics, in a database or spreadsheet, for example. Perform the following profiling activities and data capturing with the Unity Profiler:
  - CPU Usage
  - GPU Usage
  - Memory usage
  - Rendering
  - Physics

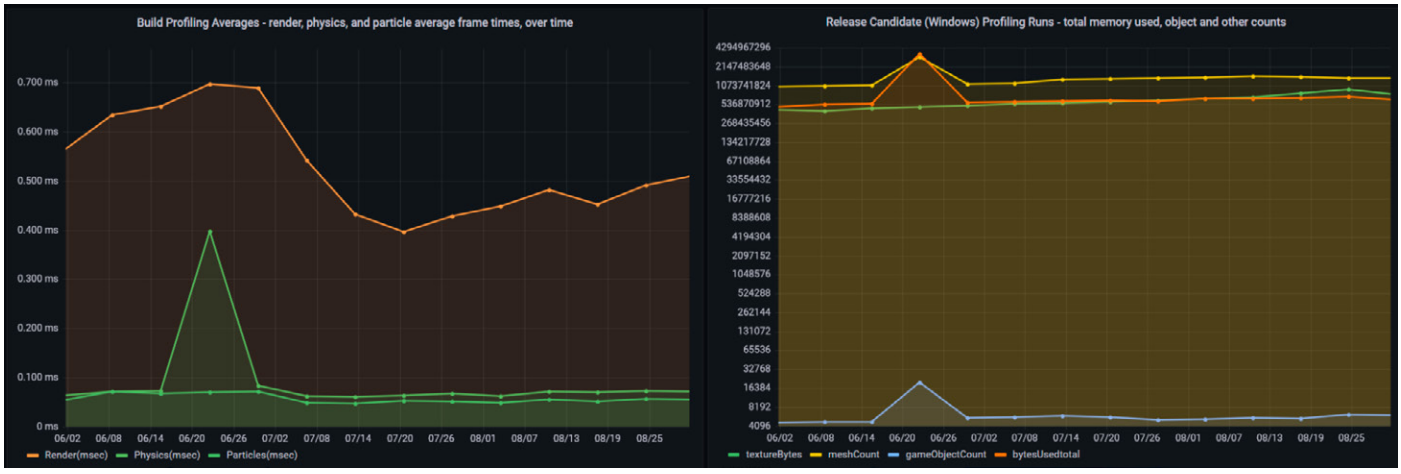
Go deeper and use these tools to record results and key difference metrics (differential against prior sprint releases):

- **Profile Analyzer**: Load previous release profiling data captures and compare and record differences.
- **Memory Profiler**: Compare prior release candidate build memory snapshots and record difference in memory increase or reduction.

Automating key performance and profiling metrics

Level up your project profiling and data capture by automating common and recurring tasks. This will save you time, and you'll benefit from metrics that are always up to date.

Metrics can be graphed and added to a project dashboard, allowing the team to see where performance has taken a nosedive (a newly added feature or bug for example), or where things have improved after an optimization and bug fixing sprint.

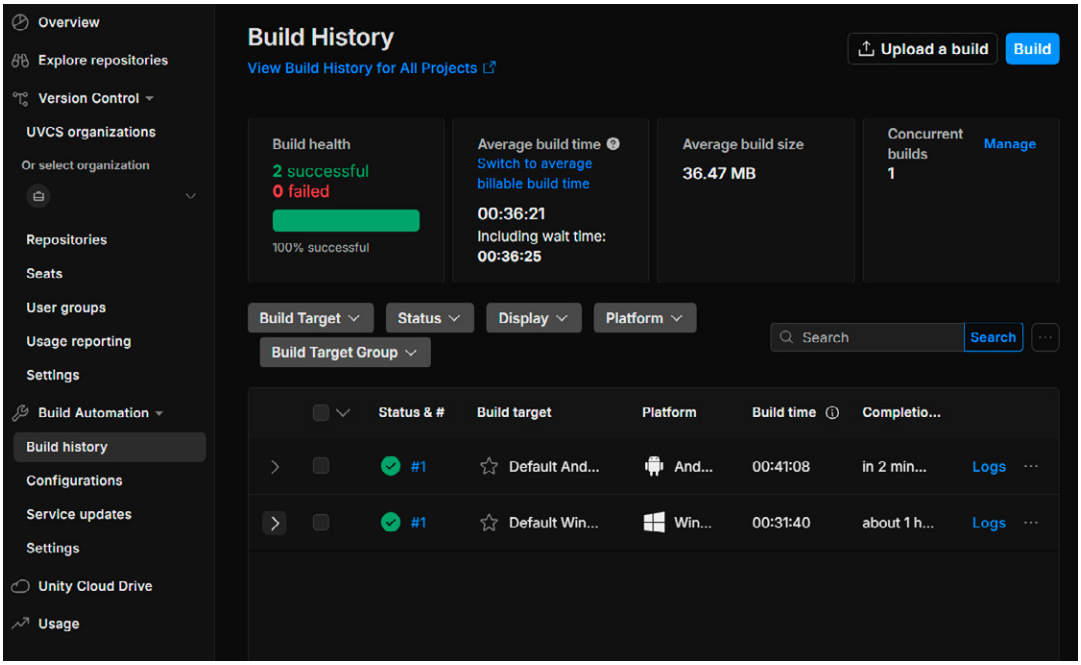


Automated weekly build profiling data captured and visualized in a Grafana dashboard

Chart a project's overall memory usage profile across all levels of the game over time. By capturing memory snapshots with the Memory Profiler and averaging the figures out across all levels, you can record a memory footprint per target device/platform, sprint, or release cycle.

If you wish to record high-level profiling statistics, use a ProfilerRecorder to record metrics such as Total Reserved Memory or System Used Memory and output these to a CI (Continuous Integration), directing them to a chart or graphing tool such as Grafana.

Use Unity DevOps tools to automate the creation of release builds and integrate this process with an automated device profiling workflow.



Check the results of your builds via the **Build Automation > Build History** feature available from the Unity Cloud dashboard. If any of them have failed, you can troubleshoot the failure by checking the logs. Learn more about project organization and Unity services in the e-book, *Best practices for project organization and version control (Unity 6 edition)*.

An automated profiling pipeline example

Automation can help ensure your team realizes the benefits of profiling builds without the worry that this process will be deprioritized due to time constraints.

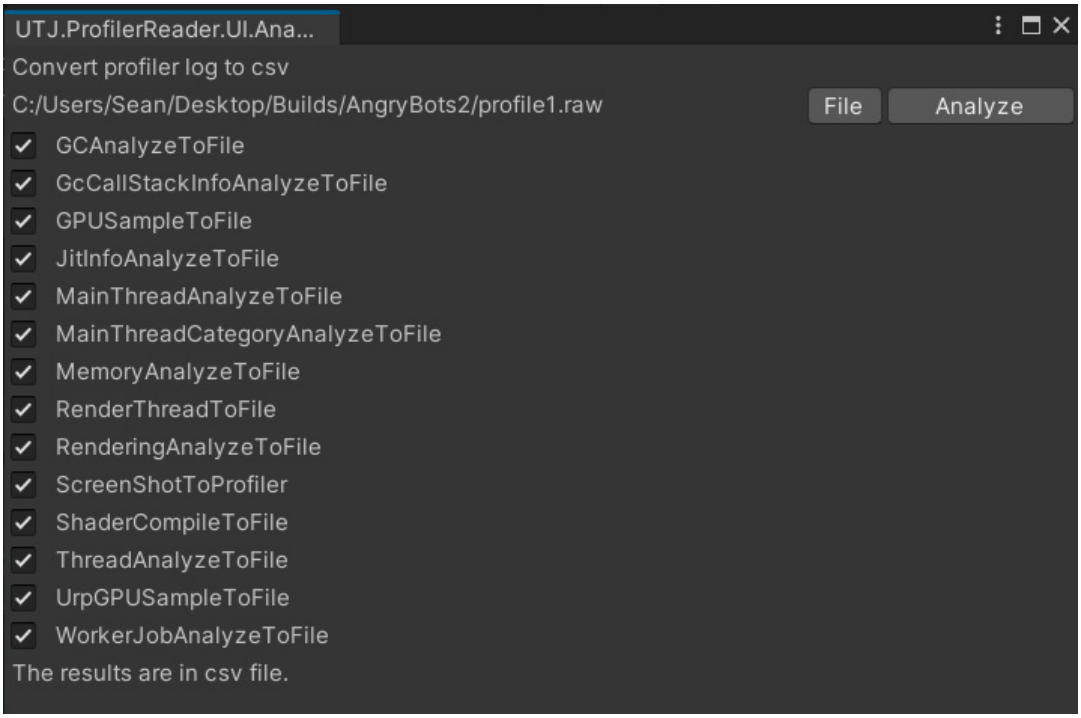
This example workflow shows how you can use automation to ensure that build profiling happens frequently and accurately.

- Use Unity Build Automation to create automated build releases.
- After each release, use a script to start a build player and capture raw profiling data over 2000 frames, e.g.:
  - **AngryBots2.exe -profiler-enable -profiler-log-file profile1.raw -profiler-capture-frame-count 2000.** To learn more about the command line arguments, check Unity documentation.

Note: Another option here would be to use a script to switch to a new log file (e.g., profile_<N+1>.raw) every 300–2000 frames, or to profile key points in an application's test cycle (checkpoints in an automated level playthrough). This stored data can then be referenced if problem areas are spotted in dashboard graphs later on.

— Profiling data is captured in the **profile1.raw** file and can now be parsed for interesting metrics.

— The next step uses ProfileReader (a tool for parsing and converting raw profile data to CSV format) to convert the raw profile data to a more readable CSV format.

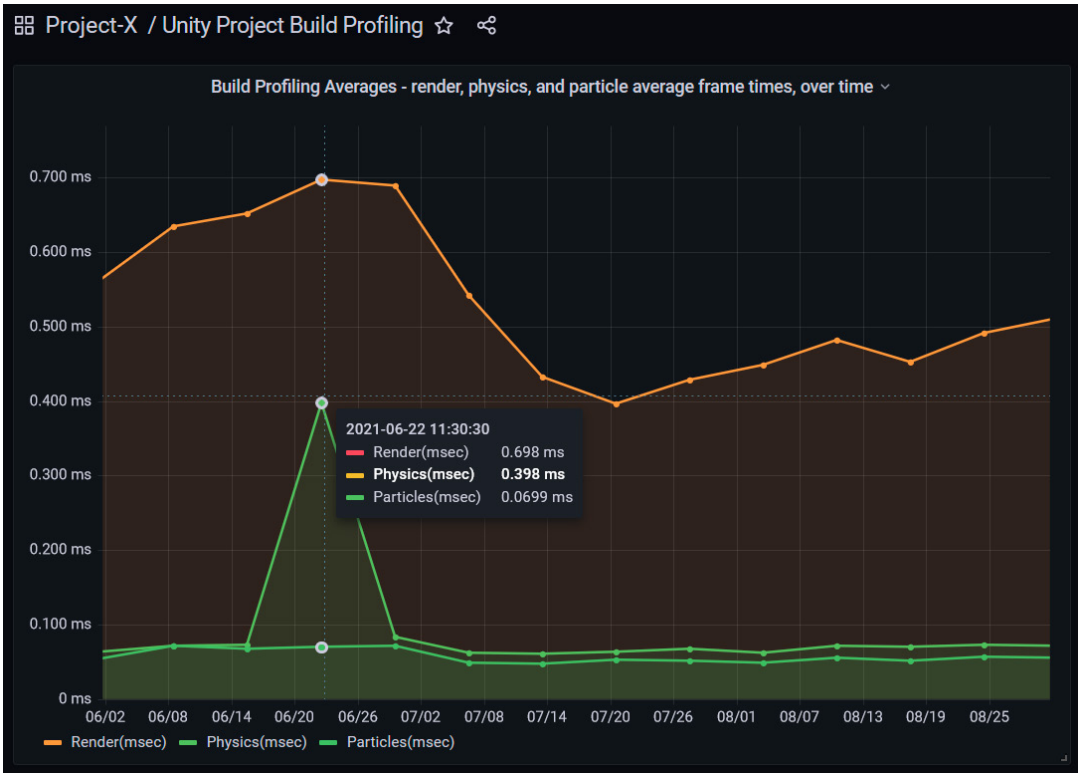


The Editor interface for ProfileReader

— ProfileReader can be used on the command line, so this stage of the pipeline would be a script to execute it:

  — Unity.exe -batchMode -projectPath "AngryBots2" -logFile .\Editor.log -executeMethod UTJ.ProfilerReader.CUIInterface.ProfilerToCsv -PH.inputFile "profile1.raw" -PH.timeout 2400 -PH.log

— With data parsed from CSV, the automated pipeline uploads data for your nightly, weekly, or sprint releases to a tool such as Grafana for visualization.



This image is of a Grafana dashboard visualizing automated profiling data. It looks like someone let a physics object creation bug creep into the build.

With data visualized and updated automatically, your team can easily spot when graphs spike to identify issues more quickly. They can also view the results of a performance optimization task or the results of the level design team doing a memory optimization pass across various levels in a game.

Performance Testing Package for Unity Test Framework

The Unity Performance Testing Package enhances the Unity Test Framework by adding tools for performance testing. It introduces APIs and test decorators that allow you to capture samples from Unity Profiler markers and custom performance metrics, both in the Editor and in built players.

In addition to measurement capabilities, the package collects configuration metadata such as build settings and hardware details, making it easier to compare results across different environments.

This package is designed to work alongside the Unity Test Framework. To use it effectively, you should be familiar with creating and running tests as outlined in the Unity Test Framework documentation.

Profiling and debugging tools index

Start your profiling with Unity's tools, and if you need greater detail, reach for the native profilers and debugging tools available for your target platform. See the index of such tools below.

Native profiling tools

Android / Arm

— Android Studio:The latest Android Studio includes a new Android Profiler that replaces the previous Android Monitor tools. Use it to gather real-time data about hardware resources on Android devices.

— Arm Performance Studio: A suite of tools to help you profile and debug your games in great detail, catered for devices running Arm hardware.

— Snapdragon Profiler: Specifically for Snapdragon chipset devices only. Analyze CPU, GPU, DSP, memory, power, thermal, and network data to help find and fix performance bottlenecks.

Intel

— Intel VTune: Quickly find and fix performance bottlenecks on Intel platforms with this suite of tools. For Intel processors only.

— Intel GPA suite: A suite of graphics focused tools to help you improve your game's performance by quickly identifying problem areas.

Xbox / PC

— PIX: PIX is a performance tuning and debugging tool for Windows and Xbox game developers using DirectX 12. It includes tools for understanding and analyzing CPU and GPU performance as well as monitoring various real-time performance counters.

PC / Universal

— AMD μProf: AMD uProf is a performance analysis tool for understanding and profiling performance for applications running on AMD hardware.

— NVIDIA NSight: Tooling that enables developers to build, debug, profile, and develop class-leading and cutting-edge software using the latest visual computing hardware from NVIDIA.

— Samply: Samply is an open source command line CPU profiler which uses the Firefox profiler as its UI. It works on macOS, Linux, and Windows.

— Superluminal: Superluminal is a high-performance, high-frequency profiler that supports profiling applications on Windows, Xbox One, and PlayStation written in C++, Rust and .NET. It is a paid product, though, and must be licensed to be used. Check out our discussions article for a quick intro on how to get started.

PlayStation

— CPU profiler tools are available for PlayStation hardware. For more details, you need to be a registered PlayStation® developer, start here.

iOS

— Xcode Instruments and the XCode Frame Debugger: Instruments is a powerful and flexible performance-analysis and testing tool that's part of the Xcode toolset.

WebGL

— Firefox Profiler: Dig into the call stacks and view flame graphs for Unity WebGL builds (among other things) with the Firefox Profiler. It also features a comparison tool to look at profiling captures side by side.

— Chrome DevTools Performance: Another web browser tool that can be used to profile Unity WebGL builds.

GPU debugging and profiling tools

While the Unity Frame Debug tool captures and illustrates draw calls that are sent from the CPU, the following tools can help show you what the GPU does when it receives those commands.

Some are platform-specific and offer closer platform integration. Take a look at the tools relevant to the platforms of interest:

— Arm Streamline: Part of Arm's Performance Studio software suite, focusing on low-overhead performance measurement of the CPU and GPU.

— Arm Frame Advisor: Part of Arm's Performance Studio software suite, focusing on frame-based API profiling.

— RenderDoc: GPU debugger for desktop and mobile platforms, focusing on frame-based API debugging.

— Intel GPA: Graphics profiling for Intel-based platforms

— Apple Frame Capture Debugging Tools: GPU debugging for Apple platforms

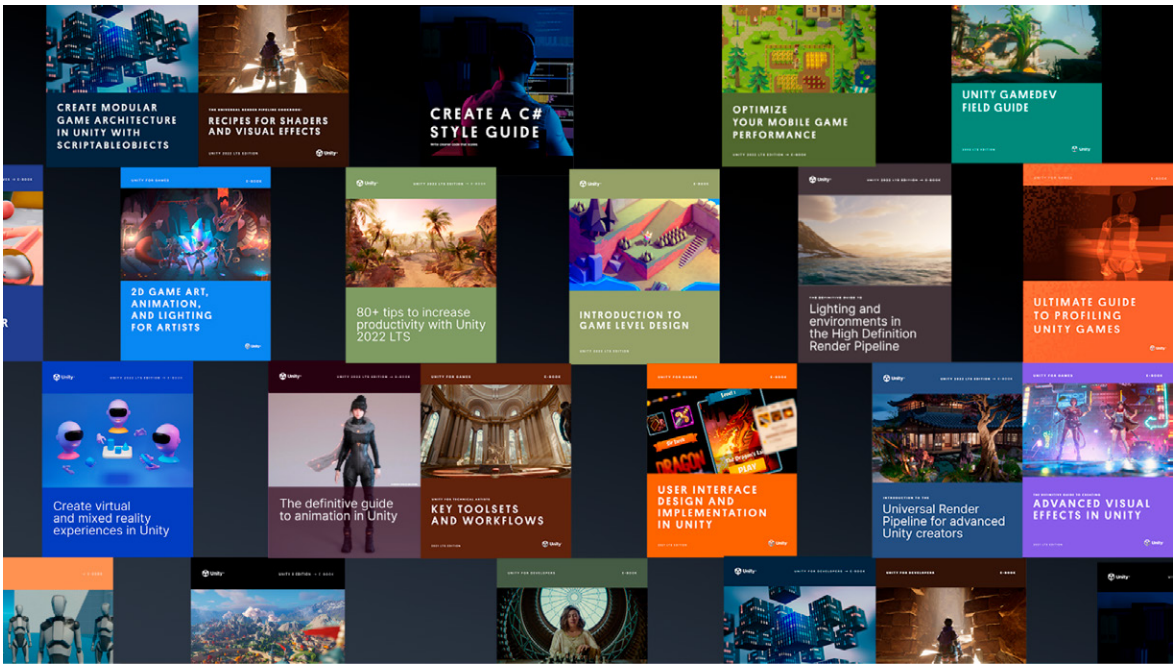
— Visual Studio Graphics Diagnostics: Choose this and/or PIX for DirectX-based platforms such as Windows or Xbox

— NVIDIA Nsight Frame Debugger: OpenGL-based frame debugger for NVIDIA GPUs

— AMD Radeon Developer Tool Suite: GPU profiler for AMD GPUs

— Xcode frame debugger: For iOS and macOS.

Resources for advanced developers and artists



You can download many more e-books for advanced Unity developers and creators from the Unity best practices hub. Choose from over 30 guides, created by industry experts, and Unity engineers and technical artists. Get best practices that will help you develop your games efficiently with Unity's toolsets and systems.

You'll also find tips, best practices, and news on the Unity Blog and Unity Discussions, as well as through Unity Learn and the **#unitytips** hashtag.


Unity®