ULTIMATE GUIDE TO PROFILING UNITY GAMES

Unity®

Contents

INTRODUCTION

Smooth performance is essential to creating immersive gaming experiences for players. By profiling and honing your game's performance for a broad range of platforms and devices, you can expand your player base and increase your chance for success.

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

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

— Profile before making major changes: Establish a baseline

— Profiling during development: Track and ensure changes don't break performance or budgets

— Profile after: Prove the changes had the desired effect

Lean, performant code and optimized memory usage lead to better performance across low- and high-end devices. Paying attention to thermal control helps precious battery cycles on mobile devices. Overall, good performance increases your players' comfort levels, which can drive higher adoption and retention.

The main author of the guide is Sean Duffy, a software engineer and game developer. Sean has developed courses, training, and tooling for professional Unity developers for over six years and coauthored books on Unity game development, including *Unity Games by Tutorials*.

Senior Unity engineers also contributed to the e-book, including Steven Cannavan, Steve McGreal, and Martin Tilo Schmitz.

Additional guides on performance optimization available from Unity include Optimize your game performance for consoles and PC and Optimize your game performance for mobile.

All the best in your profiling and optimization efforts from the Unity team.

1

PROFILING 101

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

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 of the Unity Engine.

Unity ships with a variety of profiling tools for analyzing and optimizing your code, both in the Editor and on hardware. It's also recommended to use native profiling tools for each target platform, such as those available from Arm, Apple, PlayStation, and Xbox.

Understanding profiling in Unity

Unity's profiling tools are available in the Editor and via the Package Manager.

— Unity Profiler: Measures the performance of the Unity Editor, your application in Play mode, or connect to a device running your application in development mode

— Profiling Core package: Provides APIs that you can use to add contextual information to Unity Profiler captures

— Memory Profiler: A tool that provides in-depth memory performance analysis

— Profile Analyzer: Compare two profiling datasets together to analyze how your changes affect your application's performance

The section "Unity profiling and debug tools" provides more details on how to use these tools, along with the Frame Debugger.

Sample-based vs instrumentation profiling

There are two common methods of profiling game performance:

— Sample-based profiling

— Instrumentation profiling

Sample-based profiling is when statistical data about the work that is being done in the application is collected and then analyzed.

Sample-based profilers probe the call stack every "n" nanoseconds and use call stack information to figure out when functions were called (and by which functions), as well as for how long. Accuracy with this profiling method increases by using higher sampling rate frequencies because shorter calls to functions are not missed in the call stack. However, it leads to higher overhead.

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 events for each marker. This method doesn't lose any information, but it does rely on markers being placed in order for profiling data to be captured.

The Unity Profiler is instrumentation-based. 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.

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.

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-based profiling in Unity

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.
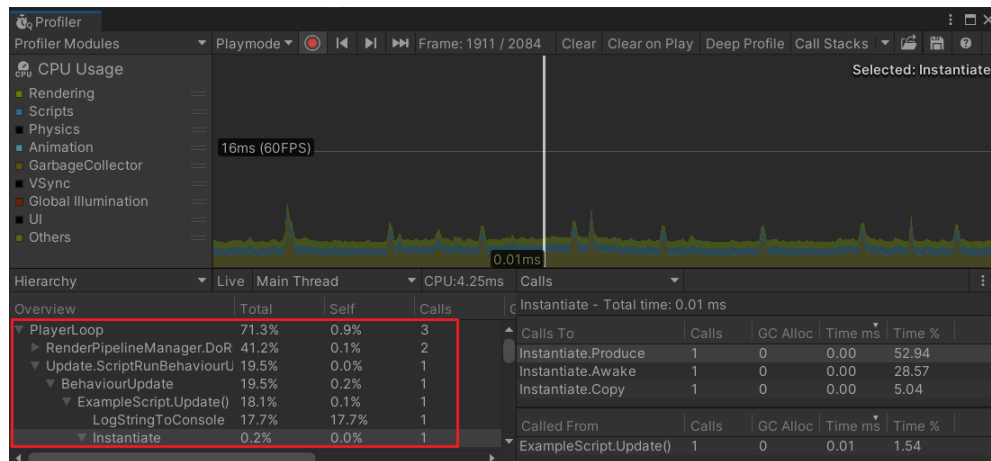
Profiling an example script shows Update() method calls to the Unity Instantiate() method

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 about them.
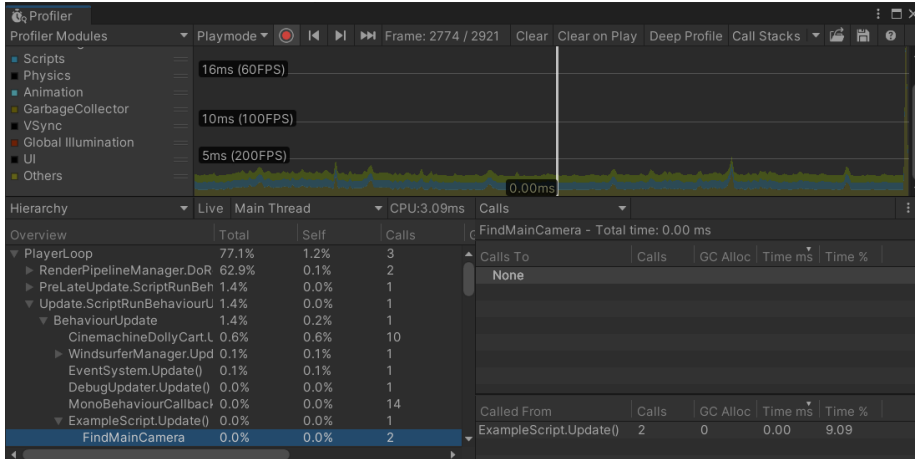
Using Camera.main results in a FindMainCamera marker appearing in a profile capture.

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 without incurring the full deep profiling overhead.

Profiler modules

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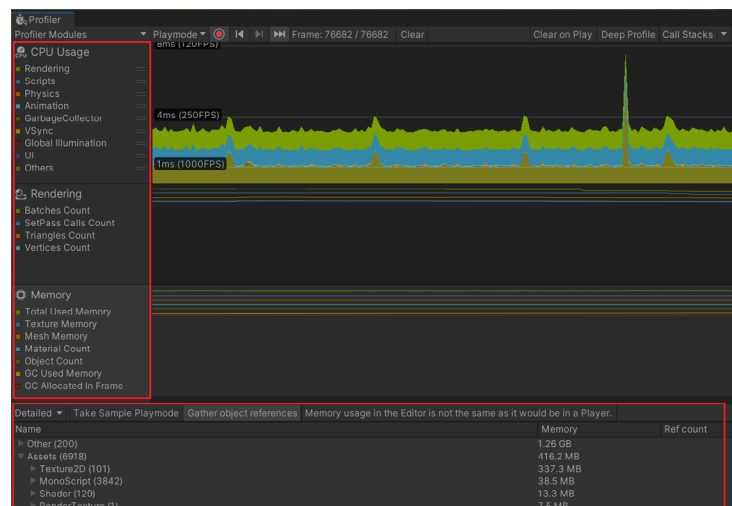
The main Profiler window view, showing the modules to the left and details panel at the bottom

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.
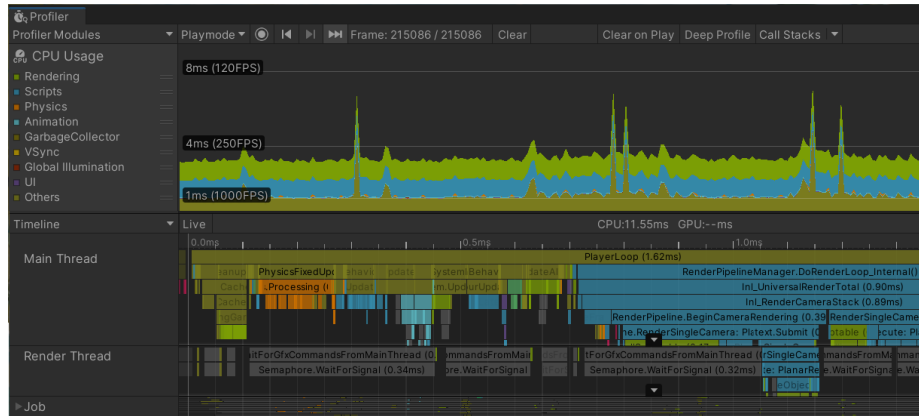
The CPU Usage module Timeline view, showing Main and Render Thread marker detail

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.

2

PROFILING WORKFLOW

This section identifies some useful goals when profiling. It also looks at common performance bottlenecks, such as being CPU-bound or GPU-bound, with details on how to identify these situations and investigate them in more detail.

Lastly, it dives into memory profiling, which is largely unrelated to runtime performance but important to know about because it can prevent game crashes.

Set a frame budget

Measuring your game's frame rate in frames per second (fps) is not ideal for delivering consistent experiences for your players. 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.

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, helping you to create 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, 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. Without it, you risk being rejected by the platform holder during your game's certification.

Frames per second: A deceptive metric

A common way that gamers measure performance is with frame rate, or frames per second. However, it's recommended that you use frame time in milliseconds instead. To understand why, look at this graph of fps versus frame time.
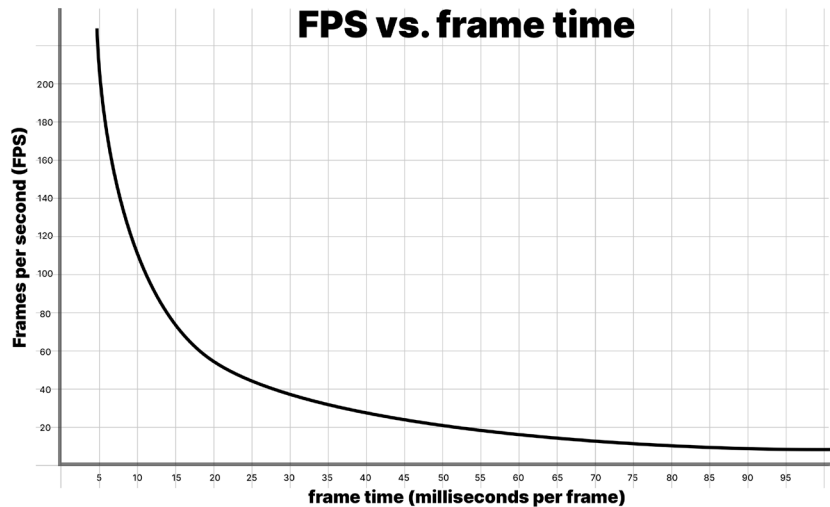
FPS vs. frame time

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 the original article, "Robert Dunlop's fps versus frame time," 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 damage to the chips (and potentially burning a player's hands!), 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 cut out 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 leveraging an early profiling technique 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

Leaving a frame idle time of around 35% is the typical recommendation to combat device thermal issues over extended play times. 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), a typical 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, most 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 (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.
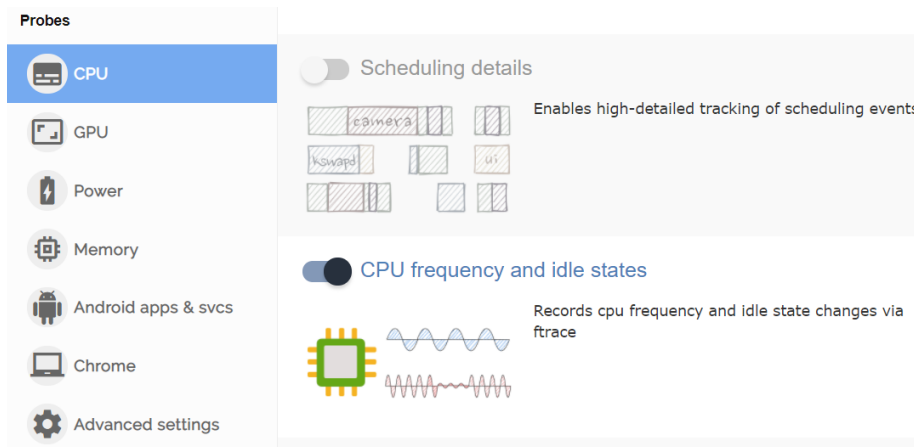
Monitor CPU frequency and idle states with tools such as FTrace or Perfetto to help identify the results of frame budget allowance optimizations.

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. Setting aside a percent of the frame budget will help to avoid these sorts of scenarios.

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 or Arm Mali hardware, Arm Mobile Studio tooling (specifically, Streamline Performance Analyzer) includes some great performance counters for identifying memory bandwidth issues. The counters are listed and explained for each Arm GPU generation, for example, Mali-G78. Note that Mobile Studio GPU profiling requires Arm Mali.
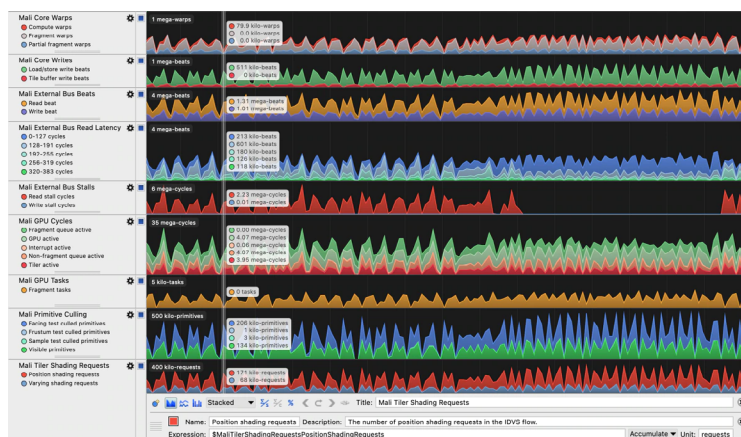
Arm's Streamline Performance Analyzer includes a wealth of performance counter information that can be captured during live profiling sessions on target Arm hardware. This is great for identifying performance issues such as memory bandwidth saturation that result from overdraw.

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 both PlayStation 4 and PlayStation 5, make sure you profile on both.

For a complete mobile optimization guide, take a look at Optimize Your Mobile Game Performance. This has many tips and tricks that will help you reduce thermal throttling and increase battery life for mobile devices running your games.

From high- to low-level profiling

A top-to-bottom approach works well when profiling, starting with Deep Profiling disabled. Use this high-level approach to collect data and take notes on which scenarios cause unwanted managed allocations or too much CPU time in your core game loop areas.

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 Locating recurring memory allocations over application lifetime section later in this guide.

If the reported call stacks are not detailed enough to track down the source of the allocations or other slowdowns, you can then perform a second profiling session with Deep Profiling enabled in order to find the source of the allocations.

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 will be affected by turning on Deep Profiling.

Read more about deep profiling further on in this guide.

Profile early

The best gains from profiling are made when you start early on in your project's development lifecycle.

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

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.

Find the bottlenecks

On some platforms, determining whether your application is CPU- or GPU-bound is easy. For example, when running an iOS game from Xcode, the fps panel shows a bar chart with the total CPU and GPU time so you can see which is the highest. Note that the CPU time includes time spent waiting for VSync, which is always enabled on mobile devices.
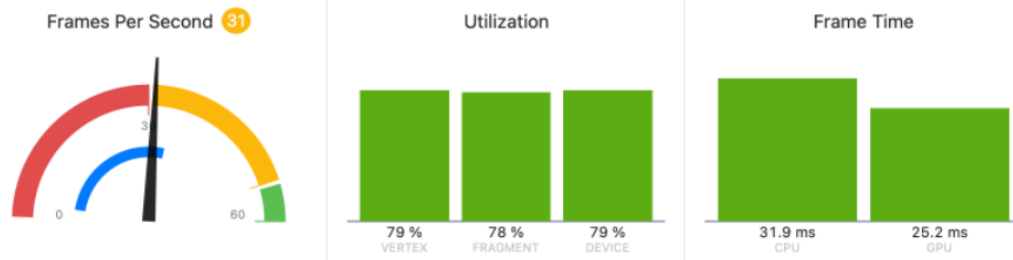
Xcode's fps view, showing a game running comfortably within 33.3 ms on both CPU and GPU

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**).

However, on some platforms it can be challenging to get GPU timing data. Fortunately, the Unity Profiler shows enough information to identify the location of performance bottlenecks. 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:

START HERE → Look at the Unity Profiler. Are we in frame budget? — **Yes** → Great. Now look at Memory!

No ↓

Is the main thread waiting for another thread? — **Yes, render thread** → Are we waiting for the GPU? — **Yes** →

No ↓ (Are we waiting for the GPU?)

No, main thread bound ↓

Yes, job threads ↓

Where is the bottleneck?

Main thread bound

Job worker thread bound

Render thread bound

GPU-bound

What now?

Implement optimizations and profile again. Compare results using Profile Analyzer to confirm performance improvements.

Use Unity Profiler and Profile Analyzer to identify biggest costs. If you can't see the problem clearly, try enabling Deep Profiling, Call Stacks, or using a native CPU profiler.

Look at Frame Debugger and Graphics or Quality settings

Look at GPU Debugger, Shaders and Graphics or Quality settings

What might the fix be?

Optimize Scripts, Physics, , Garbage alloction/collection, Cameras, UI, Animation, Job dependencies, Parallelization

Optimize Cameras, Culling, Draw call batching

Optimize Shaders, Mesh or Texture Compression or Resolution, Overdraw, Post-processing

To get a holistic picture of all CPU activity, including when it's waiting for the GPU, use the Timeline view in the CPU Usage 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 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 for some things, 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.

Below is an image of a profiling 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.
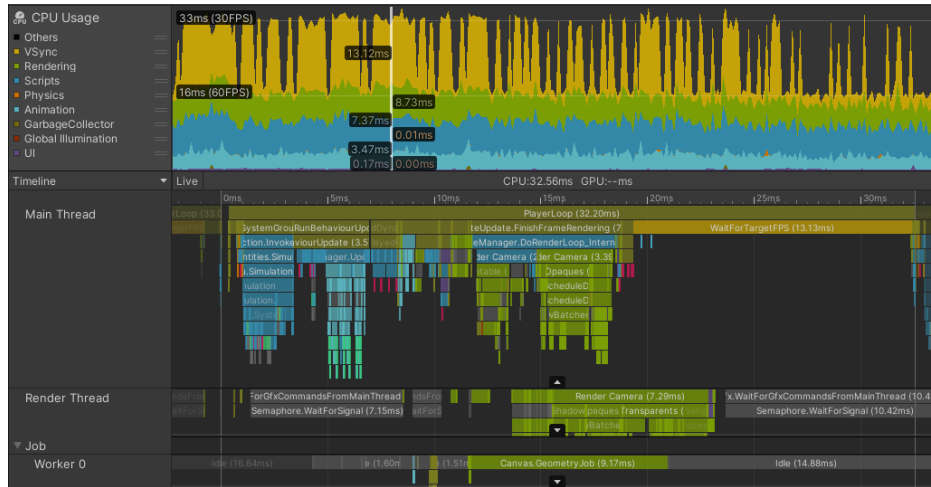
A game running comfortably within the ~22 ms frame budget required for 30 fps without overheating. Note the WaitForTargetfps padding the main thread time until VSync and the gray idle times in the render thread and worker thread. Also note that the VBlank interval can be observed by looking at the end times of Gfx.Present frame over frame, and that you can draw up a time scale in the Timeline area 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 using 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 have finished performance profiling until next time – congratulations. Consider 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. 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 on overall performance. Some "optimizations" might even worsen your game's overall performance.

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 the threads that are the most common ones for finding performance issues are:

— **The main thread:** This is where all of the game logic/scripts perform their work by default and where the majority of the time is spent for features and systems such as physics, animation, UI, and rendering.

— **The render thread:** 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 required to instruct the GPU on a particular platform.

— **The Job worker threads:** Developers can make use of the C# 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.

Main thread

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.
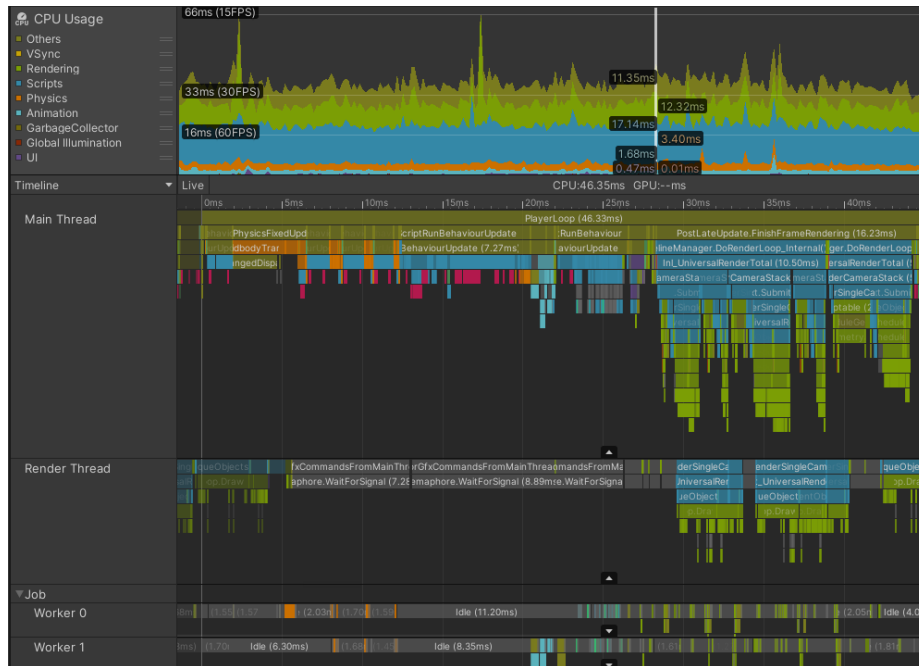
Capture from a project which is main thread bound

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 that there are five instances of a marker called Inl_ RenderCameraStack, Indicating that there are five cameras 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 marker which encompasses all MonoBehaviour Update() methods, takes 7.27 ms, and the magenta sections of the timeline indicate where scripts allocate 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 actually timed by measuring the time from a Begin to an End point. To keep their overhead small, they are recorded as just their Begin time stamp, plus the size of their allocation. The Profiler ascribes a minimal amount of time to them to make sure they are visible. 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, and 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 negatively effects on performance that are harder to measure and attribute to them directly:

— Requesting new memory from the system may affect the power budget on a mobile device, which could 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. 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

— MonoBehaviour script updates

— Garbage allocation and/or collection

— Camera culling and rendering

— Poor draw call batching

— UI updates, layouts and rebuilds

— Animation

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.

Render thread

Here's a 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.
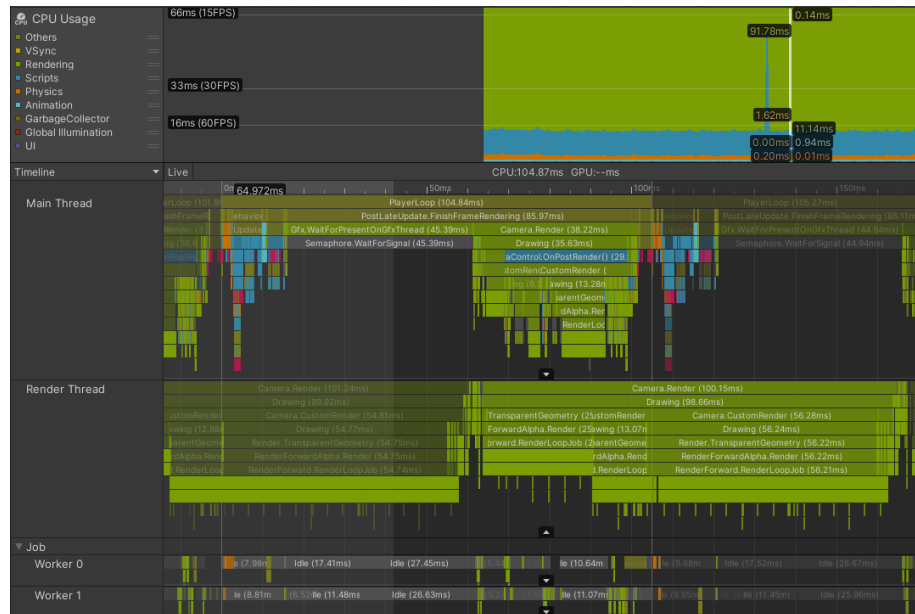
A Render thread-bound scenario

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; the render thread is 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.

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

— Poor draw call batching, 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, resulting in too many things being drawn. Investigate your Camera's frustum dimensions and cull layer masks. Consider enabling Occlusion Culling. Perhaps even create your own simple occlusion culling system based on what you know about how objects are laid out in your world. Look at how many shadow-casting objects there are in the scene – shadow culling happens in a separate pass to "regular" culling.

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.

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 C# Job System.

Here's a capture from Play mode in the Editor, showing a DOTS project running a particle fluid simulation on the CPU.
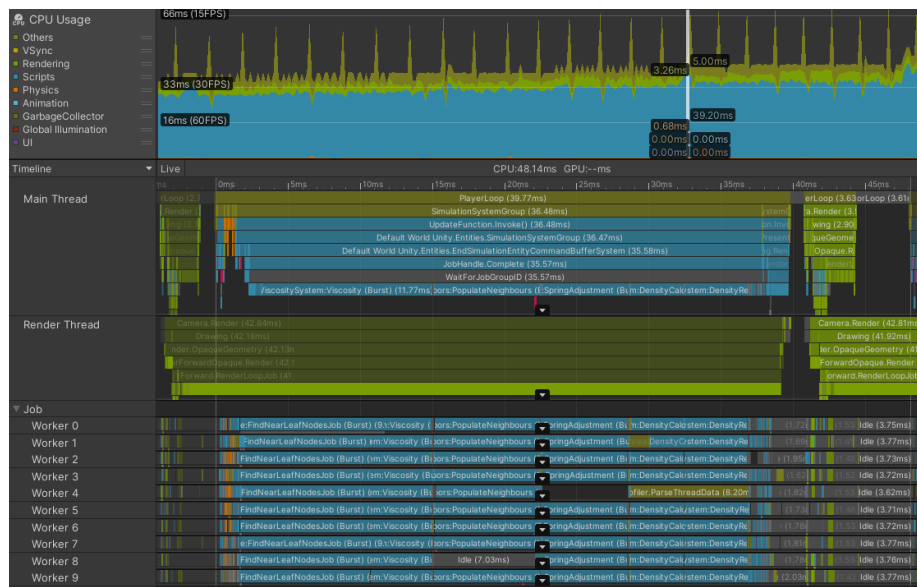
A DOTS-based project, heavy on simulation, bound by Worker threads

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 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 this guide to DOTS Best Practices.

GPU-bound

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

The following capture was taken on a Samsung Galaxy S7, 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.
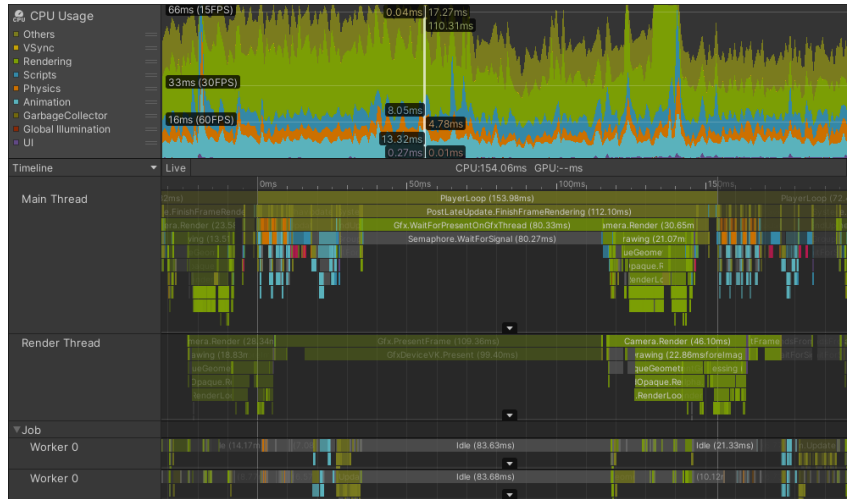
A capture from a GPU-bound mobile game

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, including common culprits like Ambient Occlusion and Bloom

— Expensive fragment shaders caused by:

— Branching logic

— Using full float precision rather than half precision

— Excessive use of registers which affect the wavefront occupancy of GPUs

— Overdraw in the Transparent render queue caused by inefficient UI, particle systems, or post-processing effects

— Excessively high screen resolutions, such as those found in 4K displays or Retina displays on mobile devices

— Micro triangles caused by dense mesh geometry or a lack of LODs, 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, or high-resolution textures without mipmaps

— Geometry or Tesselation 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.

3

MEMORY PROFILING

Memory profiling is largely unrelated to runtime performance. It's 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.

The Memory Profiler module: This is a built-in profiler module that gives you basic information on where your application uses memory.

The Memory Profiler package: This is 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.

With these built-in tools, you can monitor memory usage, locate areas of an application where memory usage is higher than expected, and find and improve memory fragmentation.

This section provides a brief introduction of memory profiling tools in Unity. For a detailed explanation of them see the Unity profiling and debug tools section.

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, 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. For example, the Xbox One console is limited to 5 GB of maximum available memory for games running in the foreground, according to 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

Each target platform has a memory limit, and once you know it, you can set a memory budget for your application. Use the Memory Profiler to look at a capture

snapshot. The Hardware Resources (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.

Hardware Resources shows the device RAM and VRAM figures the snapshot was captured on.
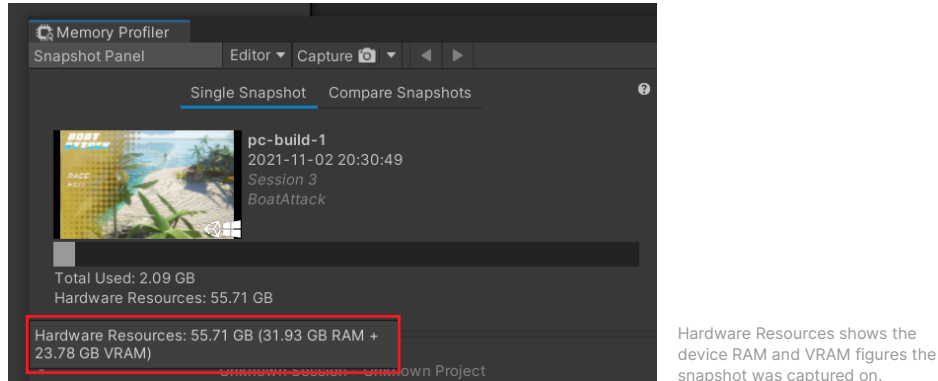
Determine the lowest specification to support for each target platform

Identify the hardware with the lowest specification in terms 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. 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.

It's important to be flexible with the budgets as the project progresses. If one team comes in way 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.

Simple and detailed views with Memory Profiler module

The Memory Profiler module provides two views: Simple and Detailed. Use the Simple view to get a high-level view of memory usage for your application. When necessary, switch to the Detailed view to drill down further.
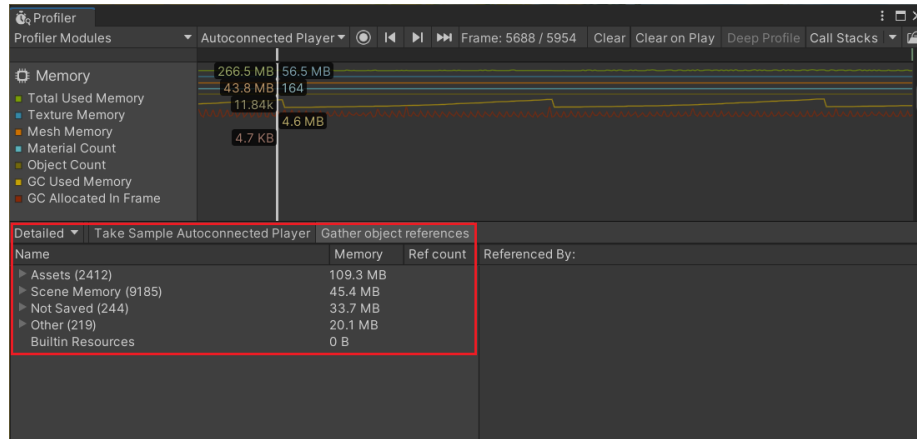
Use the Memory Profiler module to quickly gather information relating to Asset and Scene object memory allocation.

Simple

The Total Used Memory figure is the "Total Tracked by Unity Memory." It doesn't include memory that Unity has reserved (that figure is the Total Reserved Memory).

The System Used Memory figure is what the OS considers as being in use by your application. If this figure ever displays 0, be aware this indicates the Profiler counter is not implemented on the platform you are profiling. In this case, the best indicator to rely on is Total Reserved Memory. It's also recommended to switch to a native platform profiling tool for detailed memory information in these cases.

Detailed

To look into how much memory is used by your executable, DLLs, and the Mono Virtual Machine, frame-by-frame memory figures will not cut it. Use a Detailed snapshot capture to dig into this kind of a memory breakdown.
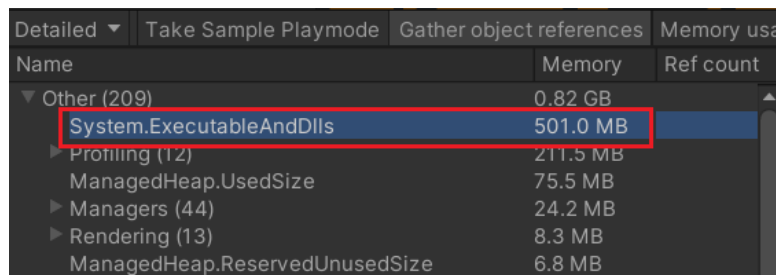
Use a captured sample to examine Detailed information such as Executable and DLL memory usage.

Note: The reference tree in the Detailed view of the Memory Profiler module only shows Native references. References from objects of types inheriting from **UnityEngine.Object** might show up with the name of their managed shells. However, they might show up only because they have Native Objects underneath them. You won't necessarily see any managed type. Let's take as an example an object with a Texture2D in one of its fields as a reference. Using this view, you won't see which field holds that reference, either. For this kind of detail. use the Memory Profiler Package.

To determine at a high level when memory usage begins to approach platform budgets, use the following "back of the napkin" calculation:

System Used Memory (or Total Reserved Memory if System Used shows 0)
+ ballpark buffer of untracked memory / Platform total memory

When this figure starts approaching 100% of your platform's memory budget, use the Memory Profiler package to figure out why.

In-depth analysis with 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 in even more detail, with visual breakdowns of your memory usage.
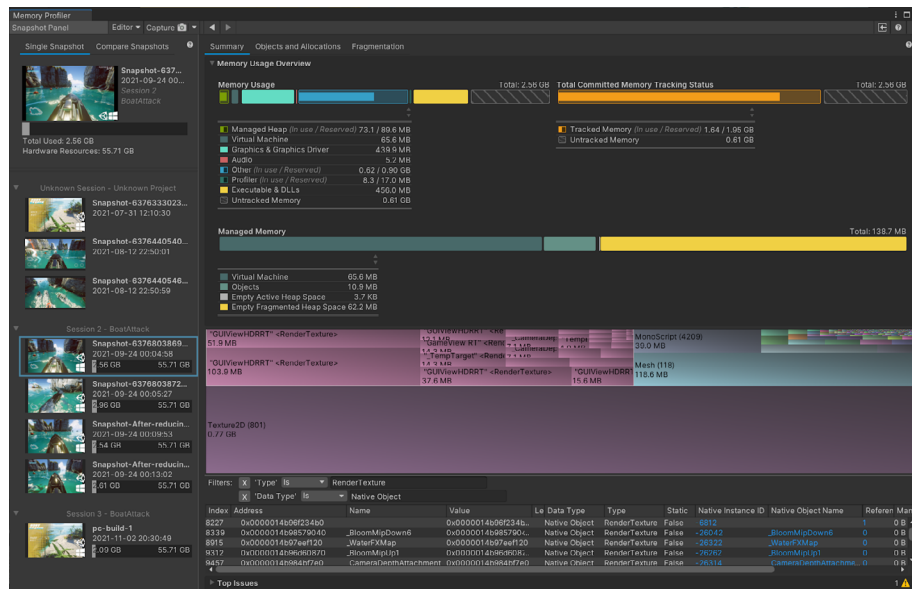
The Memory Profiler main window view

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

Alternatively, you can 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.
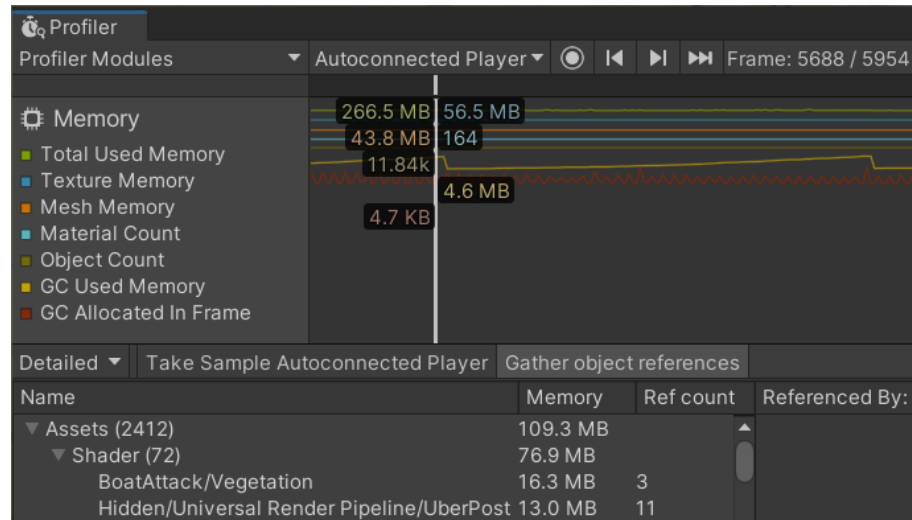
The Memory Profiler module allows you to drill down through Assets and Scene objects to easily find those with the highest utilization.

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

For example:

— **To spot GC allocations**: Although these show up in the module, they are easier to track down using Project Auditor.

— **To quickly look at the Used/Reserved size of the heap**: Newer versions of the Memory Profiler module show this information.

— **Shader memory analysis**: This is now reported in newer versions of the Memory Profiler module.

As stated earlier in this section, 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 in that 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.

Points to consider when profiling for memory usage:

— Settings such as quality levels, graphics tiers, and AssetBundle variants
   may have different memory usage on more powerful devices. For example:

   — The Quality Level and Graphics settings could affect the size of
      RenderTextures used for shadow maps.

   — Resolution scaling could affect the size of the screen buffers,
      RenderTextures and post-processing effects.

   — Texture quality setting could affect the size of all Textures.

   — The maximum LOD could affect Models and more.

   — If you have AssetBundle variants like an HD (High Definition) and an
      SD (Standard Definition) version and choose which one to use based
      on the device specifications, you also 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
      RenderTextures used for post-processing effects.

   — The supported Graphics API of a device might affect the size of Shaders
      based on which variants of them are supported or not by the API.

— Having a tiered system that uses different Quality Settings, Graphic Tier
   settings and Asset Bundle variations is a great way to be able to target
   a wider range of devices, e.g., by loading a High Definition version of an
   AssetBundle on a 4GB mobile device, and a Standard Definition 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.

4

UNITY PROFILING & DEBUG TOOLS

This section dives deeper into the capabilities of each of the profiling and debug tools available in Unity.

A note on tooling differences

Some of the tooling mentioned in this section falls under other categories such as debugging tools, for example, the Frame Debugger. While they are technically not profilers, they're important to include in your toolkit when it comes to analyzing and improving your Unity projects.

Here 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.

Profiler

The 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.

Profiling in Unity is instrumentation-based, giving you a lot of Profiler marker data to work with. Be aware that profiling directly in the Editor will add some overhead and can skew your results. Similarly, chances are your development machine may be much more powerful than your target device.

Only enable the Profiler modules you wish to work with or to use the Standalone Profiler, which provides benefits such as cleaner profiling data and reduced profiling overhead.

As a general rule of thumb, it's useful to always enable the CPU, Memory, and Renderer modules. Enable other Profiler modules such as Audio and Physics as you see fit.

Getting started with Unity profiling

Follow these steps to get started with the Unity Profiler:

— You must use a development build when profiling. Do this via **File > Build Settings > Select Development Build**.

— 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 will not need. Each module enabled will incur a performance overhead for the player. (You can observe some of this overhead using the Profiler.CollectGlobalStats marker).

— 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.
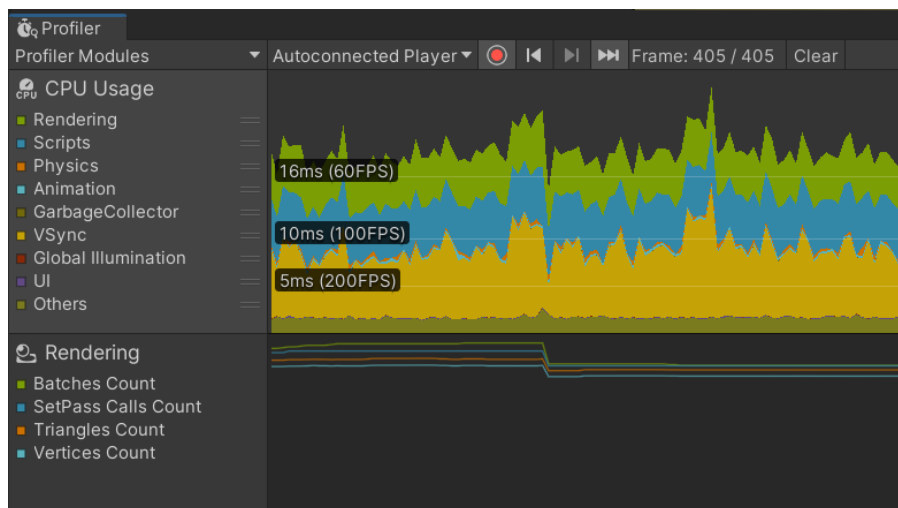
The Profiler when automatically connected to a target device

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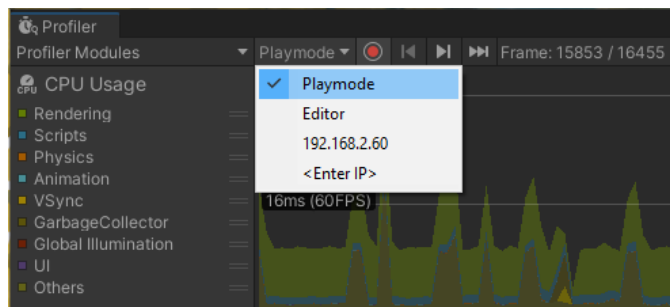
Using the Profiler to target the game running in Playmode

Profiler tips

Disable VSync and Others categories in the CPU Usage 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 markers represent 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. Go to **Edit > Project Settings…** then select **Quality** and click on the Quality Level(s) to be used on your target device and set VSync Count to Don't Sync.

Make a Development build of the game, and connect the Profiler to it. 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 Playmode or Editor mode

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

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

Using 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 and 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. Here, the Profiler launches as a new 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.

Analysis	>	Profiler	Ctrl+7
2D	>	Profiler (Standalone Process)	
AI	>	Memory Profiler	
UI Toolkit	>	Frame Debugger	
Render Pipeline	>	Physics Debugger	
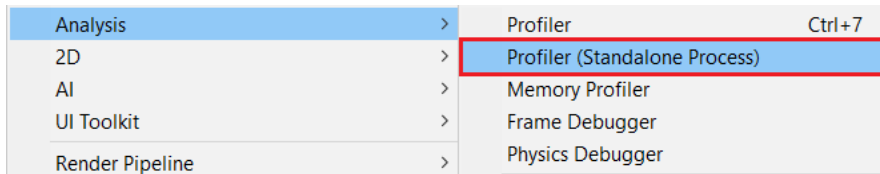
Starting the Profiler as a standalone process

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.

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

— Profiler overview in Unity manual

— Introduction to profiling in Unity

— How to profile and optimize a game

Frame Debugger

The Frame Debugger helps you to 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. The tool lets you step through the list of draw calls, one by one, so you can see the frames as they are constructed to form a scene from its graphical elements.

One advantage of the Frame Debugger over other frame debugging tools is where a draw call corresponds to the geometry of a GameObject that object will be highlighted in the main Hierarchy panel to assist identification.

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.
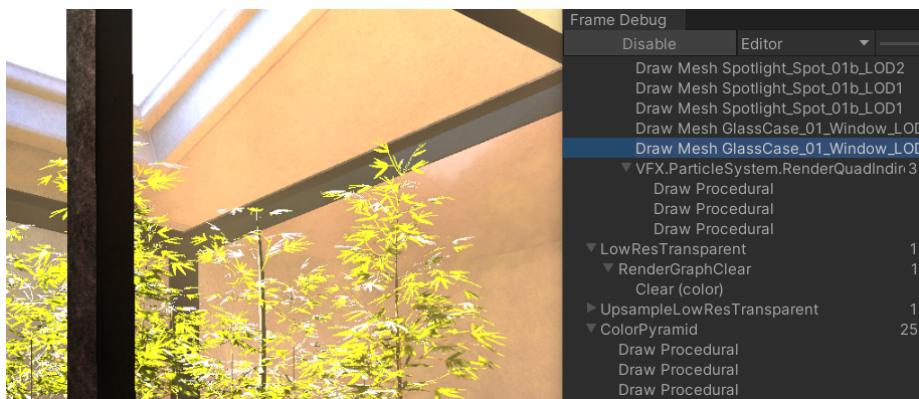
Using the Frame Debugger to analyze how identified overdraw occurs

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 list all the draw calls in sequence for the current frame on the left side of the Frame Debug window. Additional details, such as framebuffer clear events, are also included.
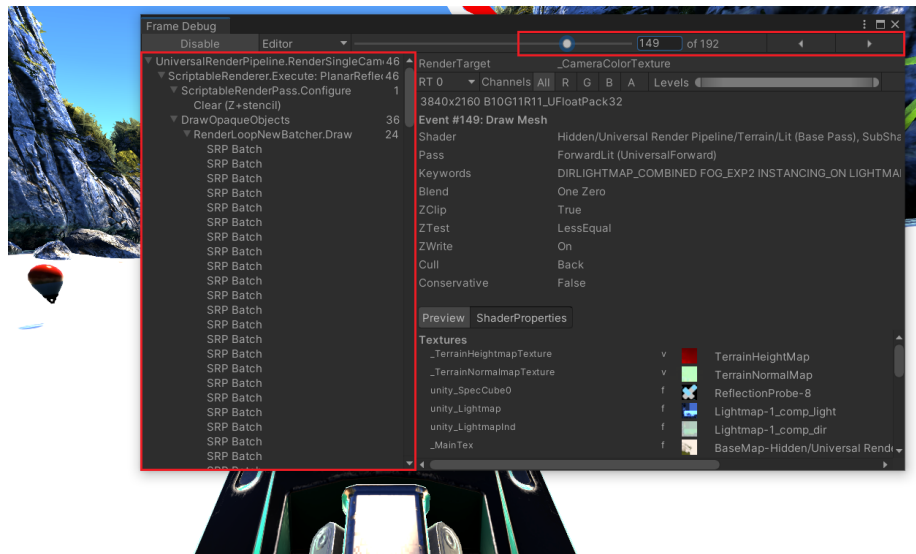
The Frame Debugger window lists draw calls and events down the left side and provides a slider to visually step through each one.

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

Unity issues draw calls 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 grouped under what's known as "render state." One way to optimize performance in this area is to reduce the number of changes to this render state.

The Frame Debugger helps identify where draw calls are coming in from. Use it to visualize and understand the rendering process to guide decisions on how to group draw calls in order to reduce changes to render state.
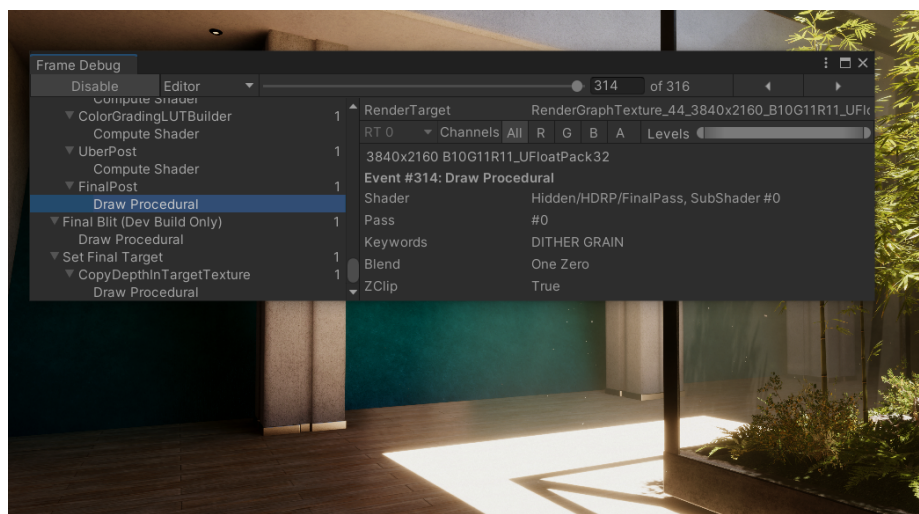
The Game window displays a scene frame constructed up to and including the selected draw call (near the end of applying post-processing effects) in the Frame Debugger.

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.

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 provided includes reasons for why a draw call could not be batched with previous ones, and an examination of the exact property values that were fed into shaders.
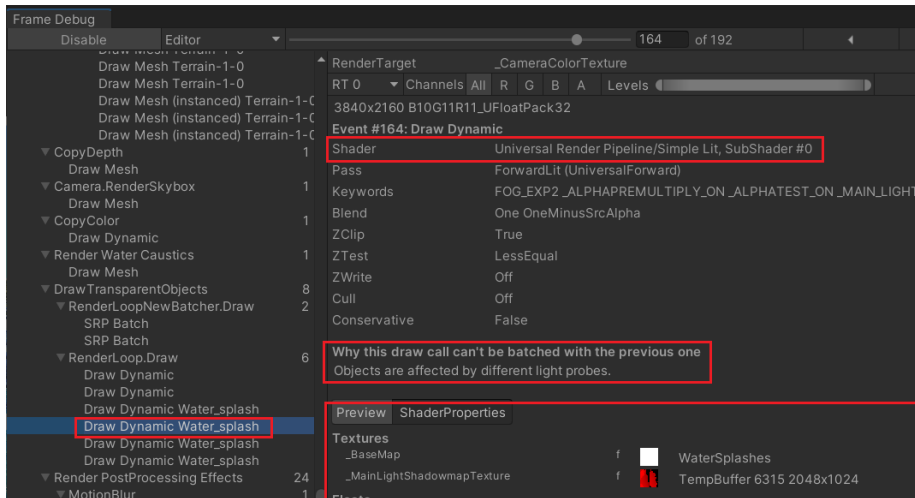
A draw call is selected and details showing its shader, reason for being excluded from batching, and shader property values are all visible in the details area.

Along with shader property values, the ShaderProperties section also reveals which shader stages it was used in (for example, **v**ertex, **f**ragment, **g**eometry, **h**ull, **d**omain).
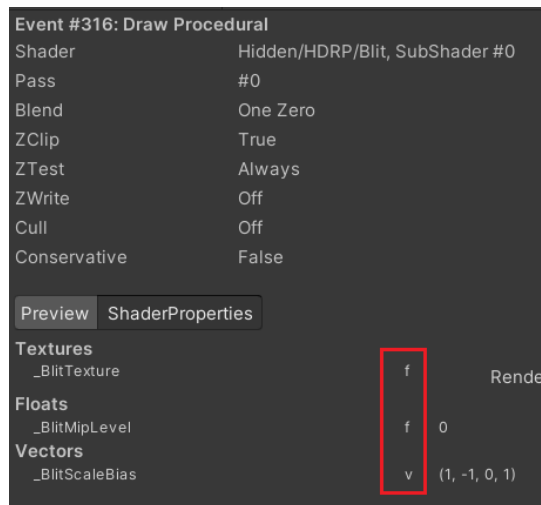
Shader stages are revealed in the ShaderProperties detail section.

Remote Frame Debugging

It's possible to attach the Frame Debugger to a player remotely on supported platforms (WebGL is not supported). For Desktop platforms, enable Run In Background for builds.

To set up remote frame debugging:

— Create a standard build of the project to your target platform (select Development Player).

— Run the player.

— Open the Frame Debug window from the Editor.

— Click the Player selection dropdown and choose the active player that is running.

— Click Enable.

You can now step through draw calls and events in the Frame Debug list hierarchy and observe the results in the active player.
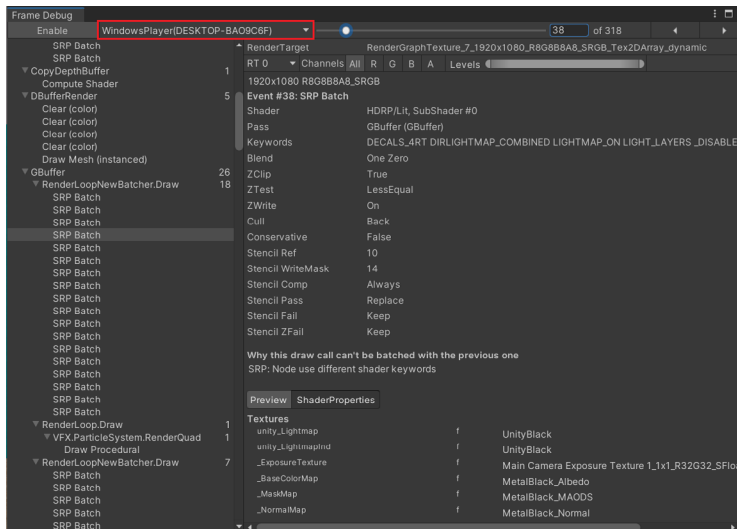
The Frame Debug window attached to a remote player build.

Render target display options

The Frame Debug window has a toolbar which lets you isolate the red, green, blue, and alpha channels for the current state of the Game view.

Isolate areas of the view according to brightness levels using the Levels slider to the right of the channel buttons. These controls are enabled when rendering into a RenderTexture.

When rendering into multiple render targets at once you can select which one to display in the Game view using the RenderTarget dropdown list.
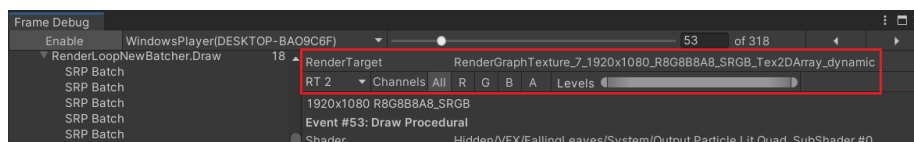
The Frame Debug render target depth channel controls.

The dropdown list also has a Depth option to show the contents of the depth buffer.
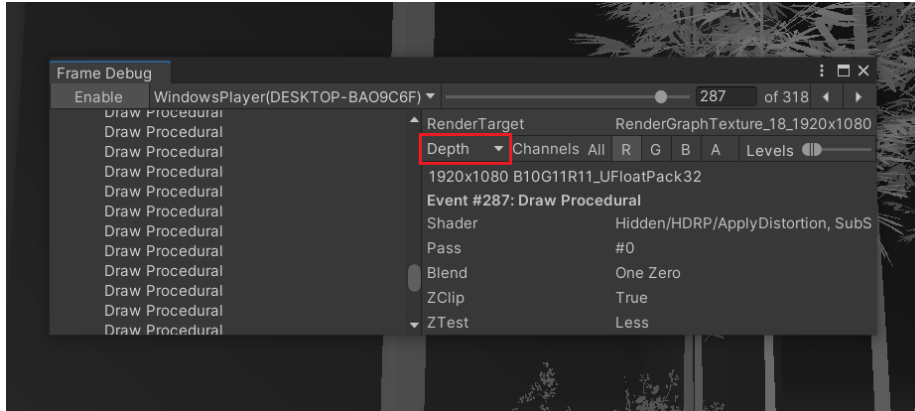
Viewing the depth buffer contents with the Frame Debug window

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 Mobile Studio for Arm hardware / 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.
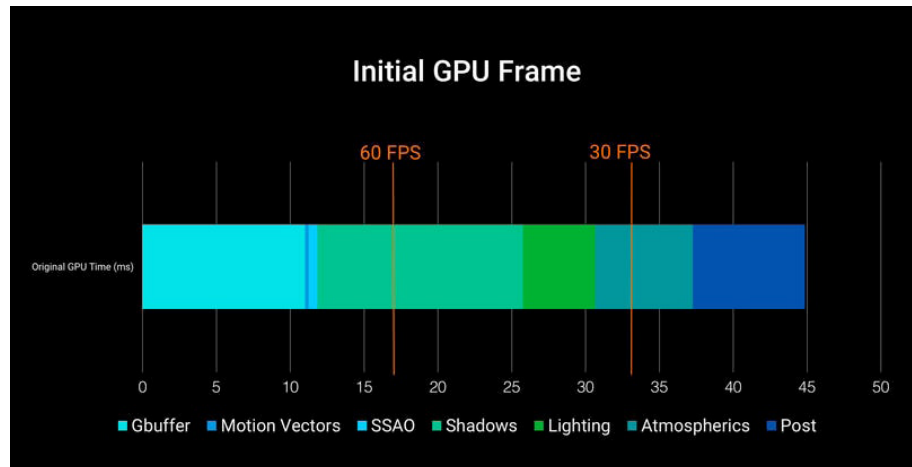
This view was GPU-bound on a PS4 Pro at roughly 45 ms per frame.

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 vital. You can achieve this with draw call batching.

Use the Frame Debugger 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: Remove objects hidden behind foreground objects and reduce overdraw. Be aware this requires additional CPU processing, so use the Profiler to ensure moving work from the GPU to CPU is beneficial.

— GPU instancing: This can reduce your batches if you have many objects that share the same mesh and material. A limited number of models in your scene can improve performance. If it's done artfully, you can build a complex scene without making it look repetitive.

— The SRP Batcher: This can reduce 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 the Universal Render Pipeline (URP) and High Definition Render Pipeline (HDRP), with as few variations between keyword combinations as possible.

Optimize fill rate by reducing 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.
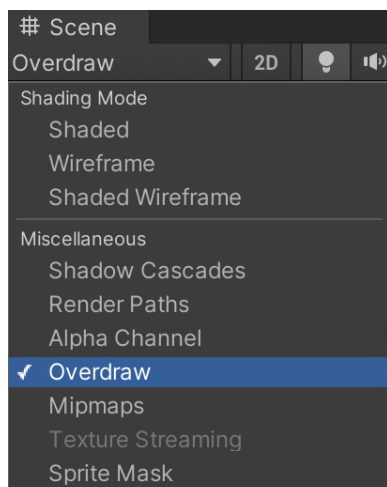
Objects rendering on top of one another create overdraw. If you're using the Built-In Render Pipeline, use the Scene view control bar to visualize overdraw. Switch the draw mode to Overdraw.

Overdraw in the Scene view control bar

Bright pixels indicate objects drawing on top of one another, while dark pixels mean less overdraw.
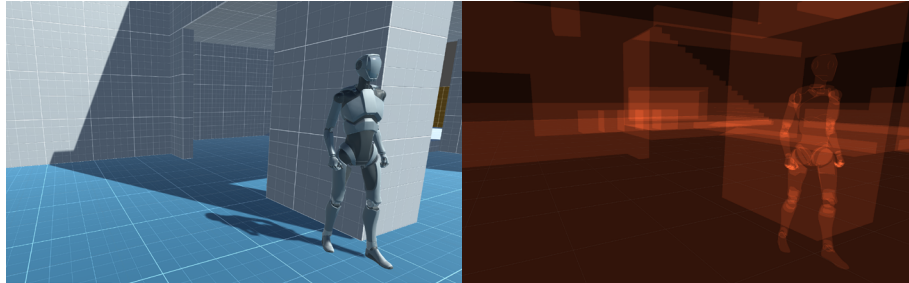


A Scene in standard Shaded view

The same Scene in Overdraw view – overlapping geometry is often a source of overdraw.

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

HDRP includes tooling that can identify overdraw. To find overdraw in your HDRP game use the Render Pipeline Debug tool via **Window > Render Pipeline > Render Pipeline Debug**.

Go to the Rendering section, and change Fullscreen Debug Mode to TransparencyOverdraw.

This debug option displays each pixel as a heat map, ranging from black, which represents no transparent pixels, through to blue and then red, the latter color indicating the Max Pixel Cost number of transparent pixels.
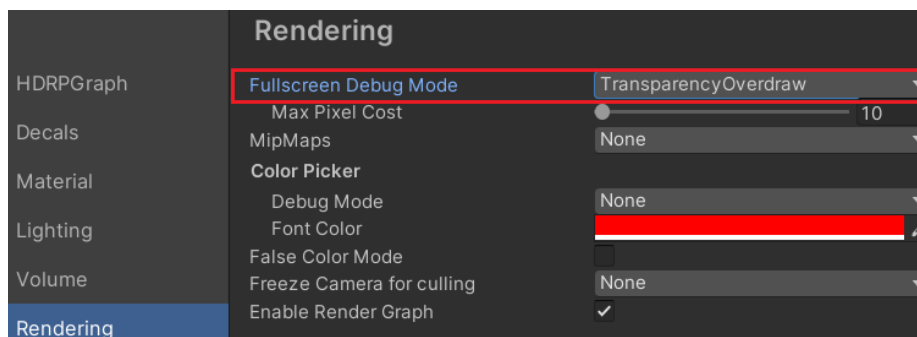
Enable TransparencyOverdraw for Fullscreen Debug Mode to help locate overdraw in scenes.

With this mode enabled, you can play through scenes and areas of your application, taking note of areas with significant overdraw.
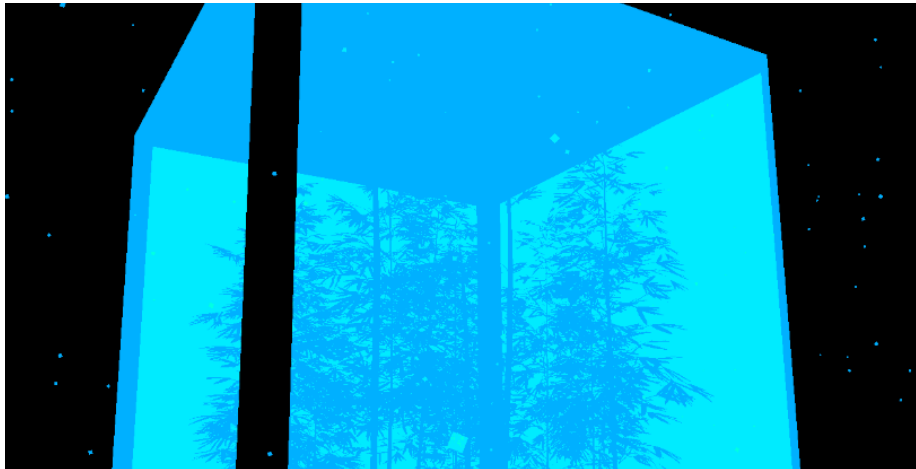
Visualizing overdraw with HDRP and the Fullscreen Debug Mode

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 in PlayStation and Xbox. Graphics Jobs allows Unity to spread the rendering work across multiple CPU cores, removing pressure from the render thread. See the 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 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

Profile Analyzer

While the standard Unity Profiler allows you to do single-frame analysis, the Profile Analyzer can aggregate and visualize profiling marker data captured from a set of Unity Profiler frames.

To get started with the Profile Analyzer:

—   Install the Profile Analyze Package via **Window > Package Manager**.

When using the Profile Analyzer, a good approach is to save profiling sessions to compare before and after performance optimization work.
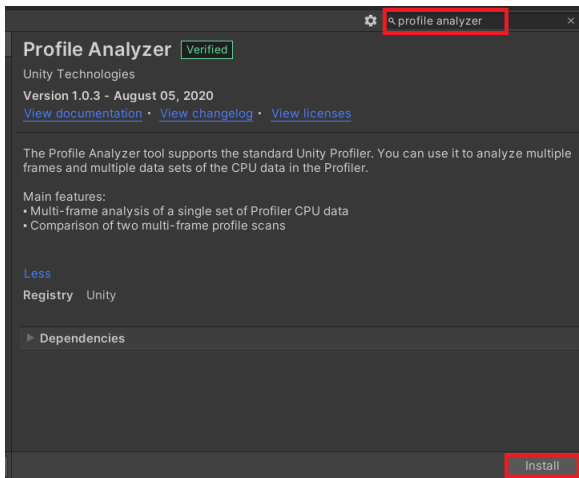
Install Profile Analyzer from the Package Manager.

Profile Analyzer pulls a set of frames captured in the Unity Profiler and performs statistical analysis on them. This data is then displayed, generating useful performance timing information for each function, such as Min, Max, Mean, and Median timings.

It can help you answer problem and optimization questions during development. Use it for A/B testing of a game scenario for performance differences, to compare before and after profiling data for code refactoring and optimization, new features, or even Unity version upgrades.
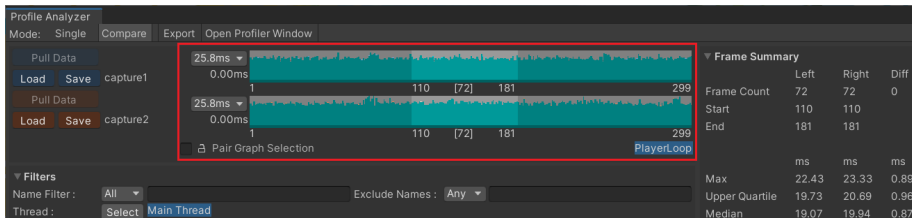
A great companion to the Unity Profiler, the Profile Analyzer can be used to aggregate and compare multiple frames captured in profiling sessions.

Using aggregated data views in the Single view of the Profile Analyzer can also answer high-level performance-over-time questions up front. This can be a better way of looking at data, 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:

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

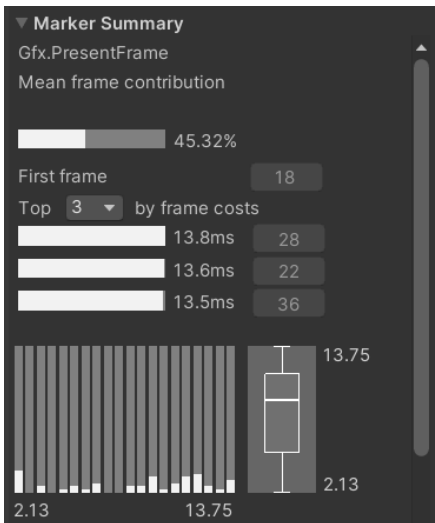— What is the mean/median/total cost of each of those markers?

The answers to such questions can be essential for pinpointing where the biggest problems are and what should be prioritized when optimizing.

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.
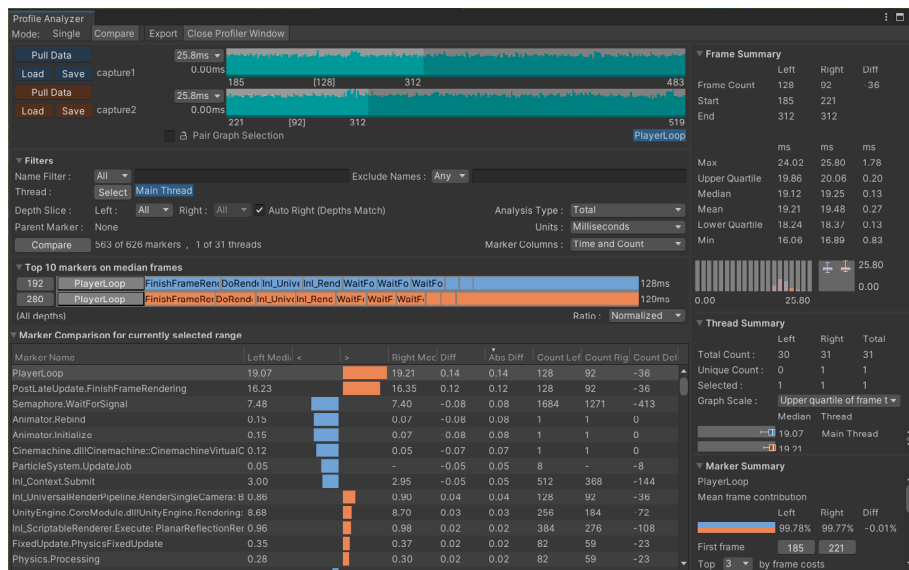
The Profile Analyzer main window overview

Profile Analyzer has multiple views and approaches for analyzing profiling data. It's divided into panels for selecting, sorting, viewing, and comparing sets of profiling data.

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

The Frame Control panel is used to select a frame or 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. 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.

Use Filters to include or exclude markers by name, or filter by specific threads. This is useful when looking through ranged selections for Time or Count statistic values.
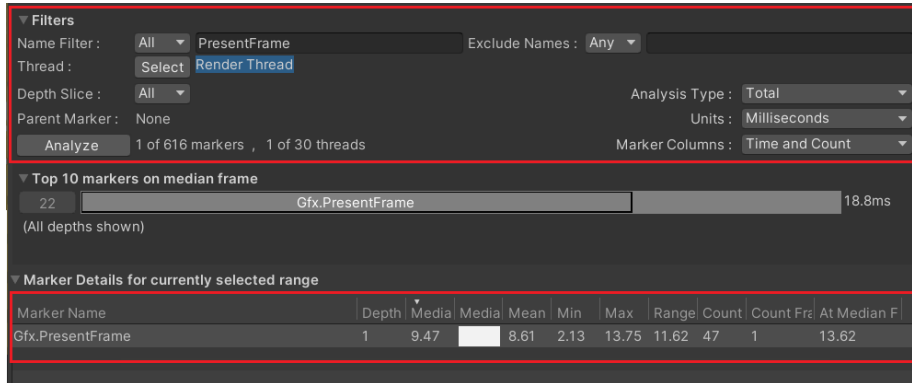
You can filter by threads or marker names to focus down on specific areas of performance data in the Marker details pane.

When adjusting filters, the Marker details pane can be customized to display different sets of statistics for your profile data. Use the Marker column dropdown to select a preset, or choose your own custom selection.
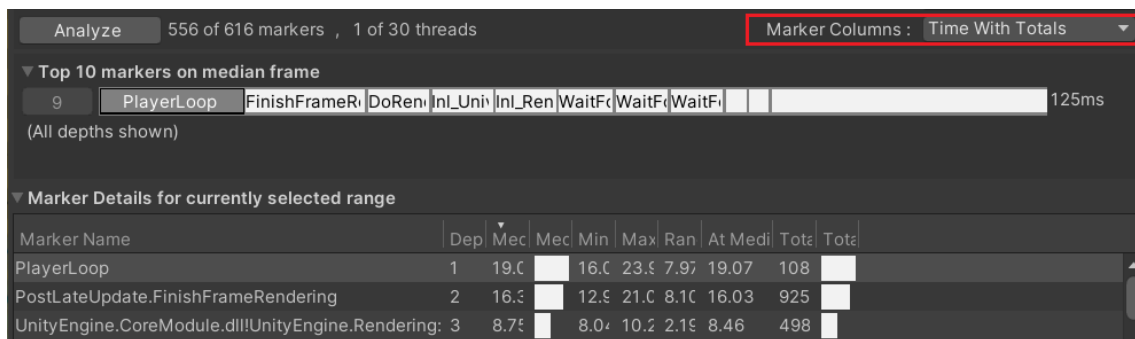
Marker column presets to customize the Marker details pane's displayed statistics

The presets are:

— **Time and count:** Displays information on the average timings and number of times the markers were called

— **Time:** Displays information on the average timings of the markers

— **Totals:** Displays information about the total amount of time the markers took on the whole data set

— **Time with totals:** Displays information about both the average and total times of the markers

— **Count totals:** Displays information about the total number of times the markers were called

— **Count per frame:** Displays information on the average total per frame the markers were called

— **Depths:** Displays information on where the markers are in the Hierarchy

— **Threads:** Displays the name of the thread that the markers appear on

Profile Analyzer views

Single view

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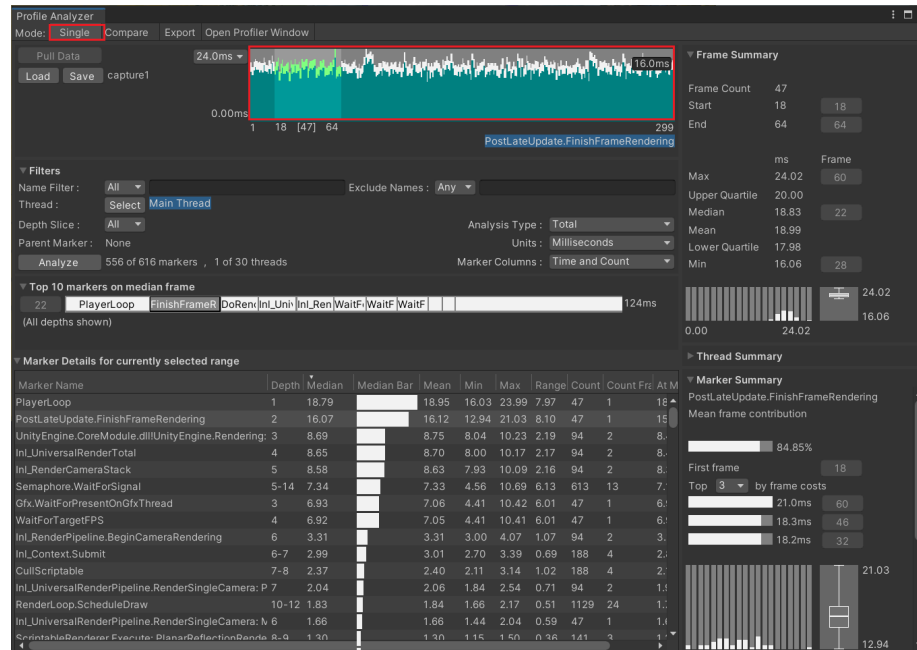
The Single view shows profile marker statistics and timings for a single or range of frames.

The Single view is an essential part of any profiling toolkit that provides many useful insights. Here are some tips for diving deeper into the data it presents.

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 and are at 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 Frame 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.

Compare view

The Compare view is where Profile Analyzer really starts to shine. In this view, you can load two data sets that the Profile Analyzer displays in two different colors.

Start by locating a specific area of your game or application to test. One way of doing this is by running a prerecorded or scripted gameplay session that can be executed multiple times. Capturing multiple session playthroughs manually also works.

Load profile session data into Profile Analyzer with the "Pull Data" method:

— Open Profile Analyzer via **Window > Analysis > Profile Analyzer**.

— Profile the deterministic session before optimization work using the Unity Profiler.

— In Profile Analyzer, switch to the Compare tab, then click the first Pull Data button to load the current capture from the Profiler.

— Apply your code and performance improvements, then clear and profile a new session again.

— Click the second Pull Data button to load the new session data.

> **Note:** If you select the Load option, the data must be in the Profile Analyzer's **.pdata** file format. If you have data from the Profiler in the **.data** file format, open it in the Profiler first, then click the Pull Data button in Profile Analyzer. Be sure to save your Profiler **.data** file before pulling it in so that you have a copy in this format, too.

Use the Marker Comparison pane to view differences in timings of markers between the first and second data sets (left and right). The columns marked with < and > show the difference if the left or right data sets are larger in value.

Marker Name	Left Median	<	>	Right Median
PlayerLoop	19.03			19.89
PostLateUpdate.FinishFrameRendering	16.13			16.63
Inl_UniversalRenderTotal	8.63			9.05
UnityEngine.CoreModule.dll!UnityEngine.Rendering:	8.68			9.10
Inl_RenderCameraStack	8.56			8.96
WaitForTargetFPS	6.91			7.08
Gfx.WaitForPresentOnGfxThread	6.92			7.09
Animator.Rebind	0.15			-
Animator.Initialize	0.15			-
Inl_RenderPipeline.BeginCameraRendering	3.28			3.42

Comparing the median and longest frames from a capture

Adjusting the Marker Columns filter will change the values that are compared accordingly.

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.
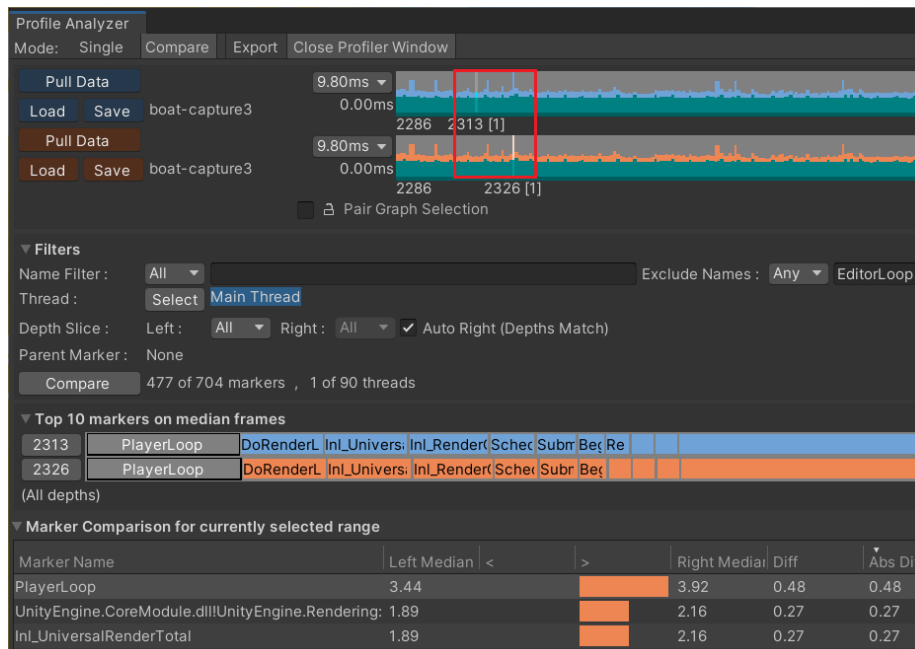
Comparing the median and longest frames from a capture

Another trick to compare data here 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).
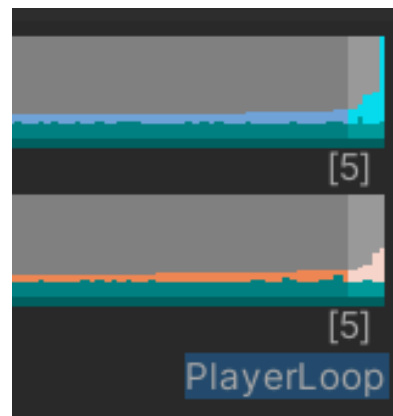
Ordering Frames by Duration and selecting an outlier range

This allows a comparison between the most ordinary and extraordinary frames. The data can then be analyzed in the filtered table called Marker Comparison for the currently selected range.

Take a look at the following resources to learn more about the Profiler Analyzer:

— CPU performance analysis with Unity's Profile Analyzer

— Introduction to profiling

Memory Profiler

The Memory Profiler is an add-on package available in the Unity Package Manager. Use the Memory Profiler to snapshot memory, either in the Editor or running in a player.

A snapshot shows memory allocations in the engine, allowing you to quickly identify the causes of excessive or unnecessary memory usage, track down memory leaks, or see heap fragmentation.

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

The Memory Profiler's top menu bar allows you to change the player selection target and capture or import snapshots.
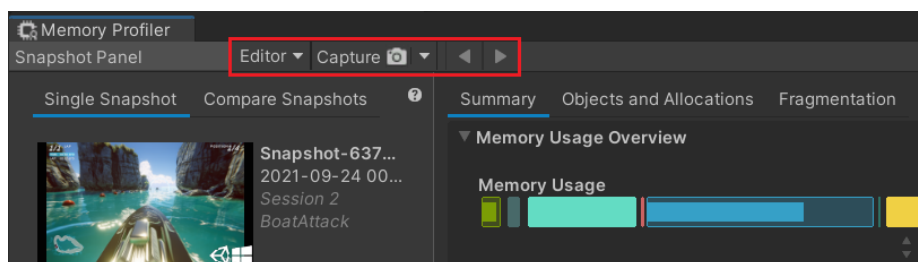
Change player selection and capture or import memory snapshots

Note: Profile memory on target hardware by connecting the Memory Profiler to the remote device with the Target selection dropdown. Profiling in the Unity Editor will give you inaccurate figures due to overheads added by the Editor and other tooling.

On the left of the Memory Profiler window is the Workbench area. Use this to manage and open or close saved memory snapshots. You can also use this area to switch between Single and Compare Snapshots views.
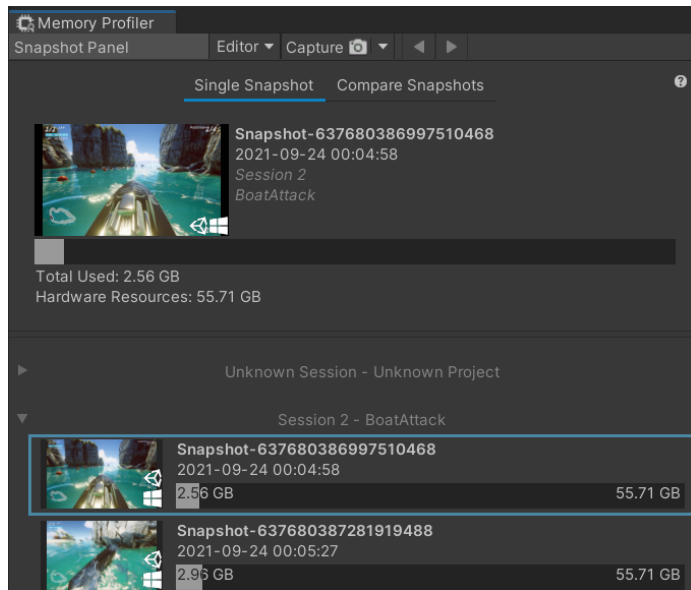
The Workbench pane is used to manage memory snapshots.

Similar to Profile Analyzer, the Memory Profiler allows you to load two data sets (memory snapshots) to compare them. This is especially useful when looking at how memory usage grew over time or between scenes and when searching for memory leaks.

Memory Profiler has a number of tabs in the main window that allow you to dig into memory snapshots including Summary, Objects and Allocations, and Fragmentation. Let's look at each of these options in detail.

The Summary view

Choose this view when you want to get a quick overview of a project's memory usage. It also contains useful and important memory related figures for the captured memory snapshot in question. It's perfect for a quick glance at what's going on at the point in time when a snapshot was taken.
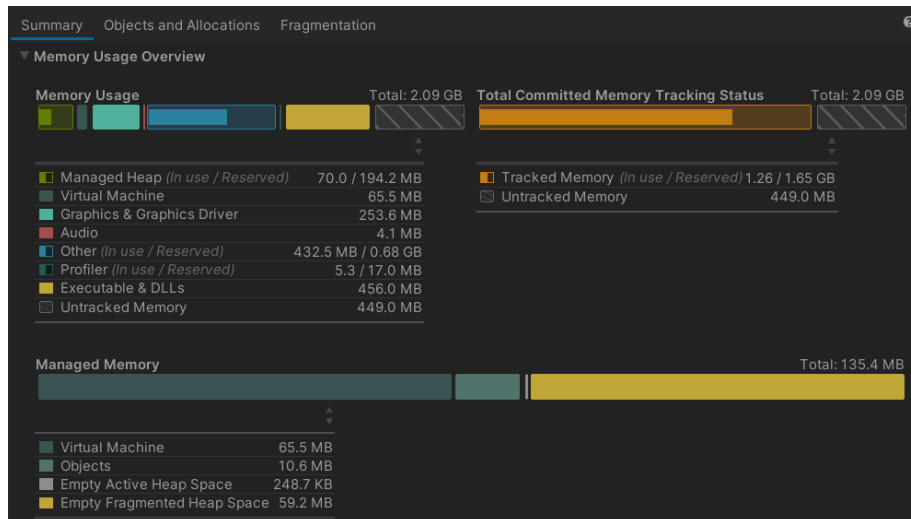
The Summary view displays an overview of memory at the time the snapshot was captured.

The Summary view also displays a breakdown of the memory usage as a graphical Tree Map that you can drill into to discover those areas that consume the most memory.

The Summary view also displays a Tree Map of memory usage for the time the snapshot was captured.

Below the Tree Map view is a filtered table that updates to display the list of objects in the selected grid cells.

The Tree Map shows memory attributed to Objects, either Native or Managed. Managed Object memory tends to be dwarfed by Native Object memory, making it harder to spot in the map view. You can zoom in on the tree map to look at these, but for inspecting smaller objects, tables usually provide a better overview. Clicking cells in the Tree Map will filter the table below it to the type of the section and/or select the specific object of interest in the Objects and Allocations view.

A filtered table of objects in the memory snapshot updates to show those in the currently selected Tree Map grid cell

You can track down which items reference objects in this list and possibly which Managed class fields these references reside in by selecting the table row or the Tree Map grid cell that represents it, then checking the References Section in the Details side panel. If the side is hidden, you can make it visible via a toggle button in the window's top right hand part of the toolbar.

The Details panel, containing References and Selection Details sections. The Reference section shows the references to the Object selected in the Tree Map or table. The Selection Details section contains details about that Object or any Object selected in the References section.

> **Note:** The Tree Map only shows Objects in memory. It's not a full representation of tracked memory. This is important to understand in case you notice that the Memory Usage Overview numbers are not the same as the Tracked Memory total.

This results from the fact that not all native memory is tied to Objects. It can also consist of non-Object-associated Native Allocations such as executables and DLLs, NativeArrays, and so on. Even more abstract concepts such as "Reserved but unused memory space" can play into the Native Allocations total.

Objects and Allocations

The Objects and Allocations view shows a table that can be switched to filter based on ready-made selections, such as All Objects, All Native Objects, All Managed Objects, All Native Allocations, and more.

The Objects and Allocations table can be filtered at many levels, allowing you to drill down into captured snapshot memory usage with high granularity.

You can switch the bottom table to display the Objects, Allocations, or Memory Regions in the selected range.

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

Load a Memory Profiler snapshot and go through the Tree Map view 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 view, locate Texture objects, Meshes, AudioClips, RenderTextures, shader variants, and preallocated buffers. These are all good candidates for memory optimization.

Locating memory leaks

A memory leak typically happens when:

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

— An object stays in memory because of an unintentional reference

The Memory Profiler Diff view can help find memory leaks by comparing two snapshots over a specific timeframe.

A common memory leak scenario in Unity games can occur after unloading a scene.

The Memory Profiler package has a workflow that guides you through the process of discovering these types of leaks using the Diff view.

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.

Memory Profiler module

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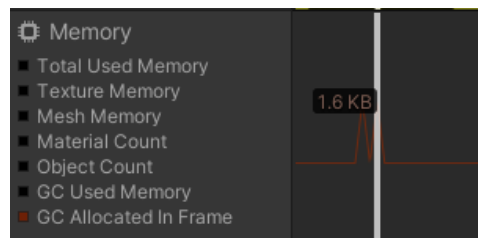
Any spikes seen for GC Allocated In Frame give you pointers to investigate for managed allocations.

Timeline view in the CPU Usage Profiler module

The Timeline view in the CPU Usage Profiler module shows allocations, including managed ones, in pink, making them easy to see and hone in on.
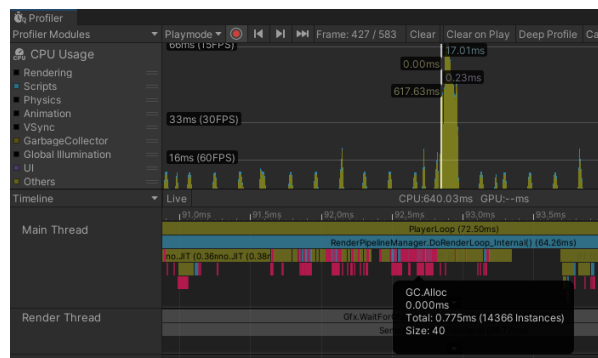
Managed allocations appear as pink-colored markers in the Timeline view.

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.

Enabling Allocation call stacks in the Profiler will allow you to follow the call stack back to the source for managed allocations.

The Related Data panel in the Hierarchy view will also reveal allocation call stack details.
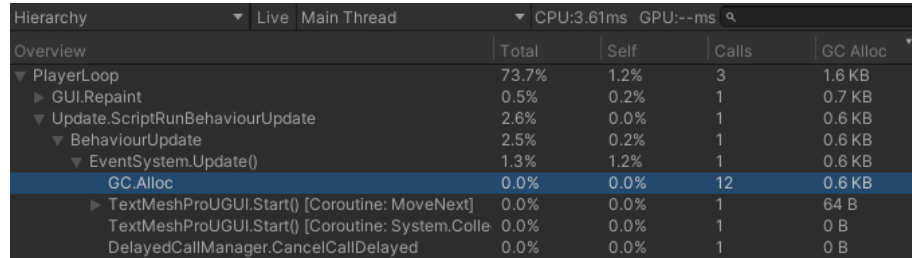
GC.Alloc samples selected in the Hierarchy or Raw Hierarchy will now contain their call stacks. You can also see the call stacks of GC.Alloc samples in the selection tool-tip in Timeline.

The Hierarchy view in the CPU Usage Profiler module

The Hierarchy view in the CPU Usage Profiler module 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.

Hierarchy	Live	Main Thread		CPU:3.61ms	GPU:--ms 🔍	
Overview			Total	Self	Calls	GC Alloc
▼ PlayerLoop			73.7%	1.2%	3	1.6 KB
▶ GUI.Repaint			0.5%	0.2%	1	0.7 KB
▼ Update.ScriptRunBehaviourUpdate			2.6%	0.0%	1	0.6 KB
▼ BehaviourUpdate			2.5%	0.2%	1	0.6 KB
▼ EventSystem.Update()			1.3%	1.2%	1	0.6 KB
GC.Alloc			0.0%	0.0%	12	0.6 KB
▶ TextMeshProUGUI.Start() [Coroutine: MoveNext]			0.0%	0.0%	1	64 B
TextMeshProUGUI.Start() [Coroutine: System.Colle			0.0%	0.0%	1	0 B
DelayedCallManager.CancelCallDelayed			0.0%	0.0%	1	0 B

Using the Hierarchy view in the CPU Usage Profiler module is a great way to filter and focus on managed allocations.

Project Auditor

Project Auditor is an experimental static analysis tool. It does a lot of useful things, several of which are outside the scope of this guide, but it can produce a list of every single line of code in a project which causes a managed allocation, without ever having to run the project. It's a very efficient way to find and investigate these sorts of issues.

Memory and GC optimizations

Reduce the impact of garbage collection (GC)

Unity uses the Boehm-Demers-Weiser garbage collector, which stops running your program code and only resumes normal execution once its work is complete.

Be aware of unnecessary heap allocations that can cause GC spikes.

— **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 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).

— **Boxing:** 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.

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 can reduce 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.

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

— Memory Profiler documentation

— Improve memory usage with the Memory Profiler in Unity tutorial

— Memory Profiler: The Tool for Troubleshooting Memory-related Issues Unite session

— Working with the Memory Profiler Unity Learn session

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 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.

Hierarchy	▼	Live	Main Thread	▼	CPU:12.58ms GPU:--ms	🔍		
Overview							Total ▼	Calls
▼ PlayerLoop							81.8%	3
▼ Update.ScriptRunDelayedDynamicFrameRate							43.1%	1
▼ CoroutinesDelayedCalls							43.1%	1
▼ SetupCoroutine.InvokeMoveNext()							36.9%	1542
▶ <WeMustGoDeeper>d__19.MoveNext()							32.0%	1541
▼ <GoDeep>d__18.MoveNext()							1.3%	1
▶ Debug.Log()							1.0%	1
▶ String.Format()							0.2%	1
▶ MonoBehaviour.StartCoroutine()							0.0%	2
▶ Transform.get_position()							0.0%	2
Component.get_transform()							0.0%	3
▼ BikeBehaviours.WeMustGoDeeper()							0.0%	1
Object.wbarrier_conc()							0.0%	1
GC.Alloc							0.0%	1
<WeMustGoDeeper>d__19..ctor()							0.0%	1

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

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 > Deep Profiling Support.**
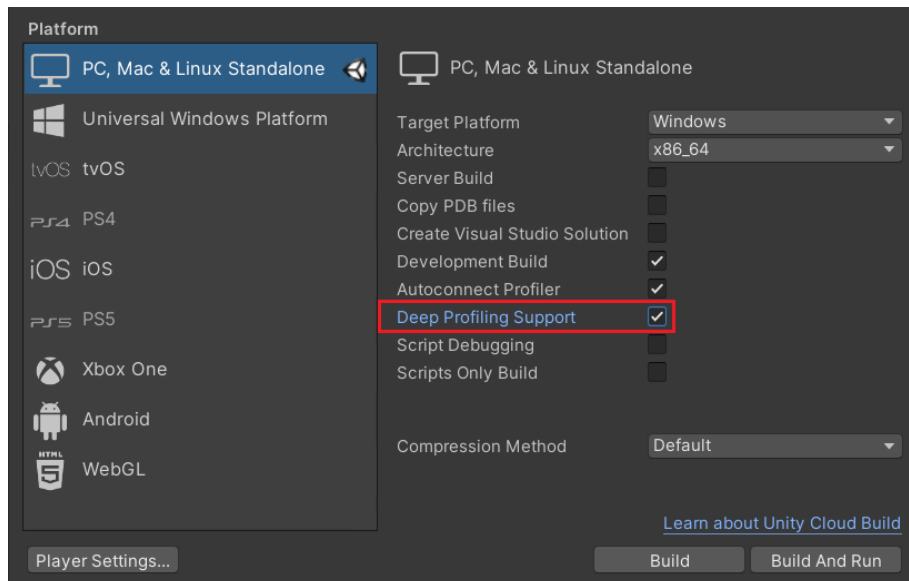
Enabling Deep Profiling Support

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

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

Hierarchy ▼	Live	Main Thread	▼	CPU:6.91ms	GPU:

Overview	Total	Self	Calls
▼ PlayerLoop	99.9%	0.5%	1
▶ TimeUpdate.WaitForLastPresentationAndUpdat	74.7%	0.0%	1
▼ PostLateUpdate.FinishFrameRendering	19.0%	0.1%	1
▼ RenderPipelineManager.DoRenderLoop_Inter	18.5%	0.0%	1
▶ RenderPipeline.InternalRender()	18.4%	0.0%	1
▶ RenderPipelineManager.GetCameras()	0.0%	0.0%	1
▶ RenderPipelineManager.PrepareRenderPip	0.0%	0.0%	1
▼ Array.Clear()	0.0%	0.0%	2
Array.ClearInternal()	0.0%	0.0%	2
ScriptableRenderContext..ctor()	0.0%	0.0%	1
RenderPipelineManager.get_currentPipelin	0.0%	0.0%	2
▼ UIEvents.CanvasManagerRenderOverlays	0.1%	0.0%	1
▼ UGUI.Rendering.RenderOverlays	0.1%	0.0%	1
▼ Canvas.RenderOverlays	0.0%	0.0%	1
▶ Material.SetPassFast	0.0%	0.0%	1
Transform.GetScene	0.0%	0.0%	1
▶ WatermarkRender	0.0%	0.0%	1

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 absolutely 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 Unity 2019.3 and onward allows 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.
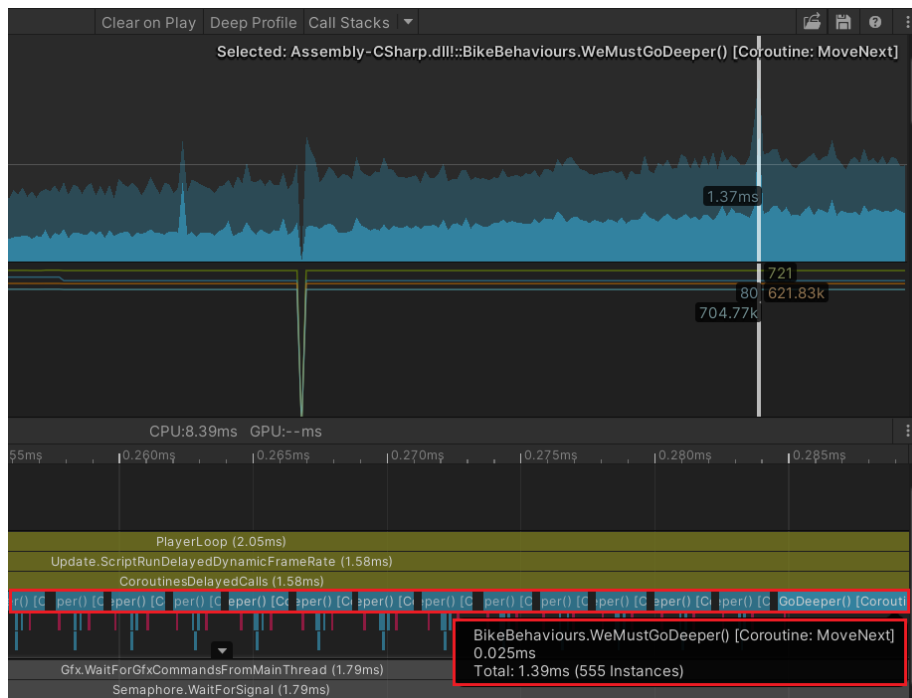
Add Profiler markers to profile deeper layers of code when deep profiling adds too much overhead.

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 on-screen, 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
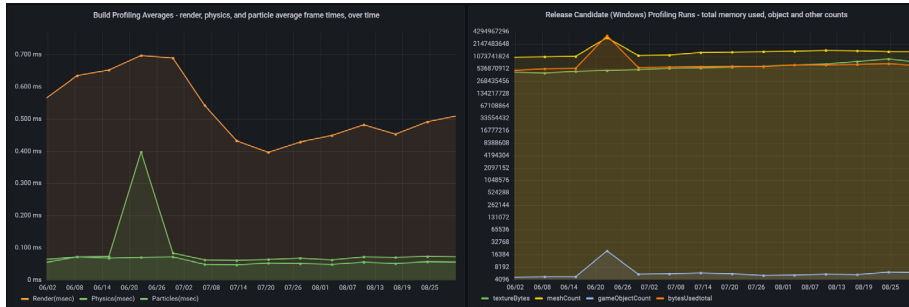
Automated weekly build profiling data captured and visualized in a Grafana dashboard

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.

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.
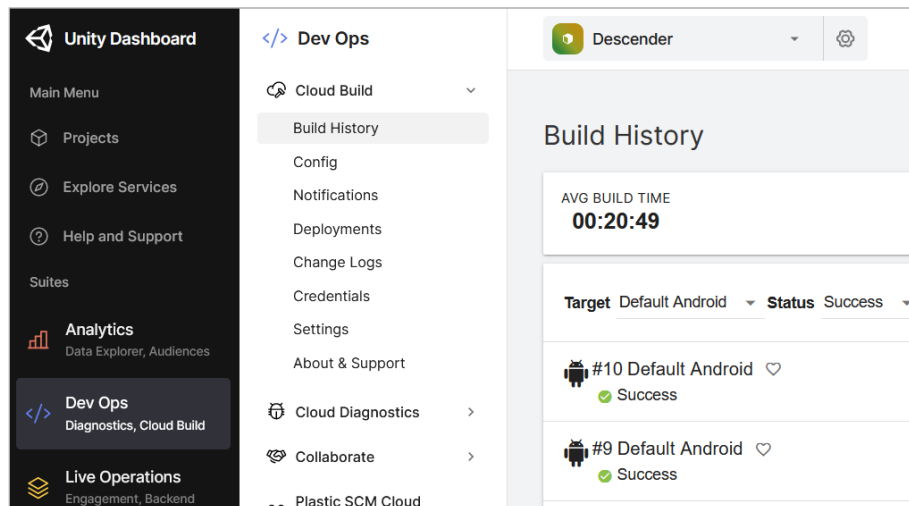
Use Unity DevOps tools such as Cloud Build to automate profiling workflows.

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 such as Cloud Build to automate the creation of release builds and integrate this process with an automated device profiling workflow.
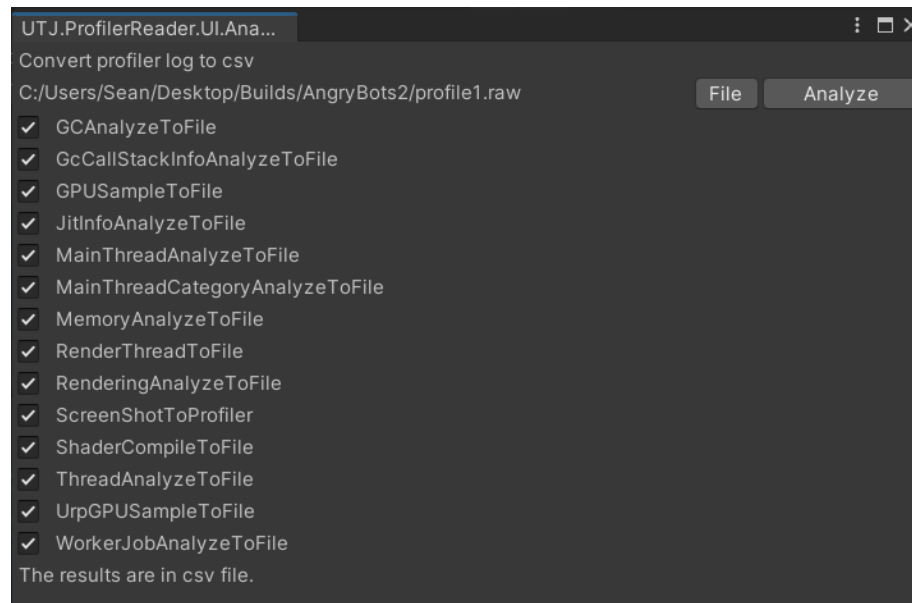
An automated profiling pipeline example

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 Cloud Build to create automated build releases.

— After each release, use a script to start a built player and capture raw profiling data over 2000 frames:

    — **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.

| UTJ.ProfilerReader.UI.Ana…	⋮ □ ✕
Convert profiler log to csv	
C:/Users/Sean/Desktop/Builds/AngryBots2/profile1.raw	File Analyze
✔ GCAnalyzeToFile	
✔ GcCallStackInfoAnalyzeToFile	
✔ GPUSampleToFile	
✔ JitInfoAnalyzeToFile	
✔ MainThreadAnalyzeToFile	
✔ MainThreadCategoryAnalyzeToFile	
✔ MemoryAnalyzeToFile	
✔ RenderThreadToFile	
✔ RenderingAnalyzeToFile	
✔ ScreenShotToProfiler	
✔ ShaderCompileToFile	
✔ ThreadAnalyzeToFile	
✔ UrpGPUSampleToFile	
✔ WorkerJobAnalyzeToFile	
The results are in csv file.	

The Editor interface for ProfileReader

— 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.

— 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**
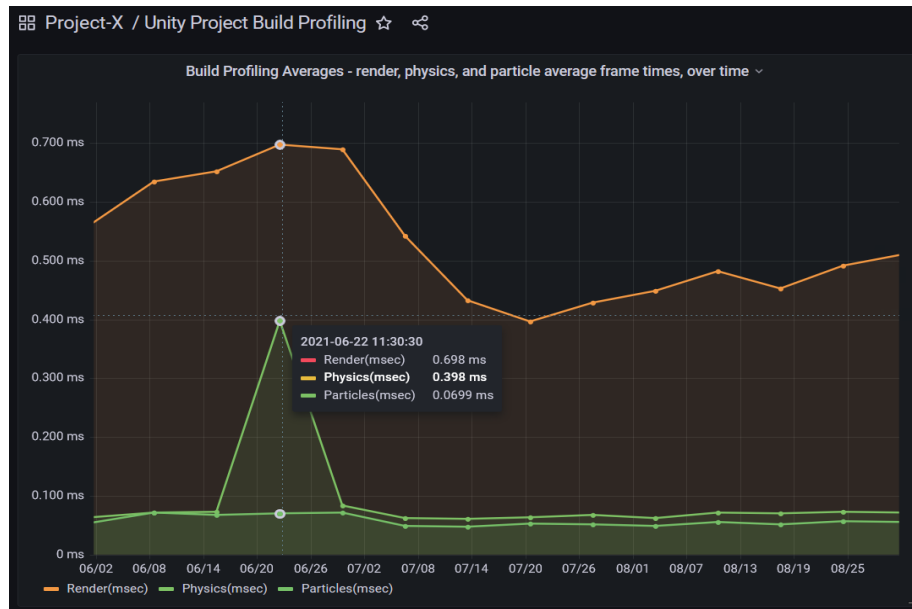
Visualizing automated profiling data in a Grafana dashboard. In this screenshot, it looks like someone let a physics object creation bug creep into the build.

— 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.

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.

5
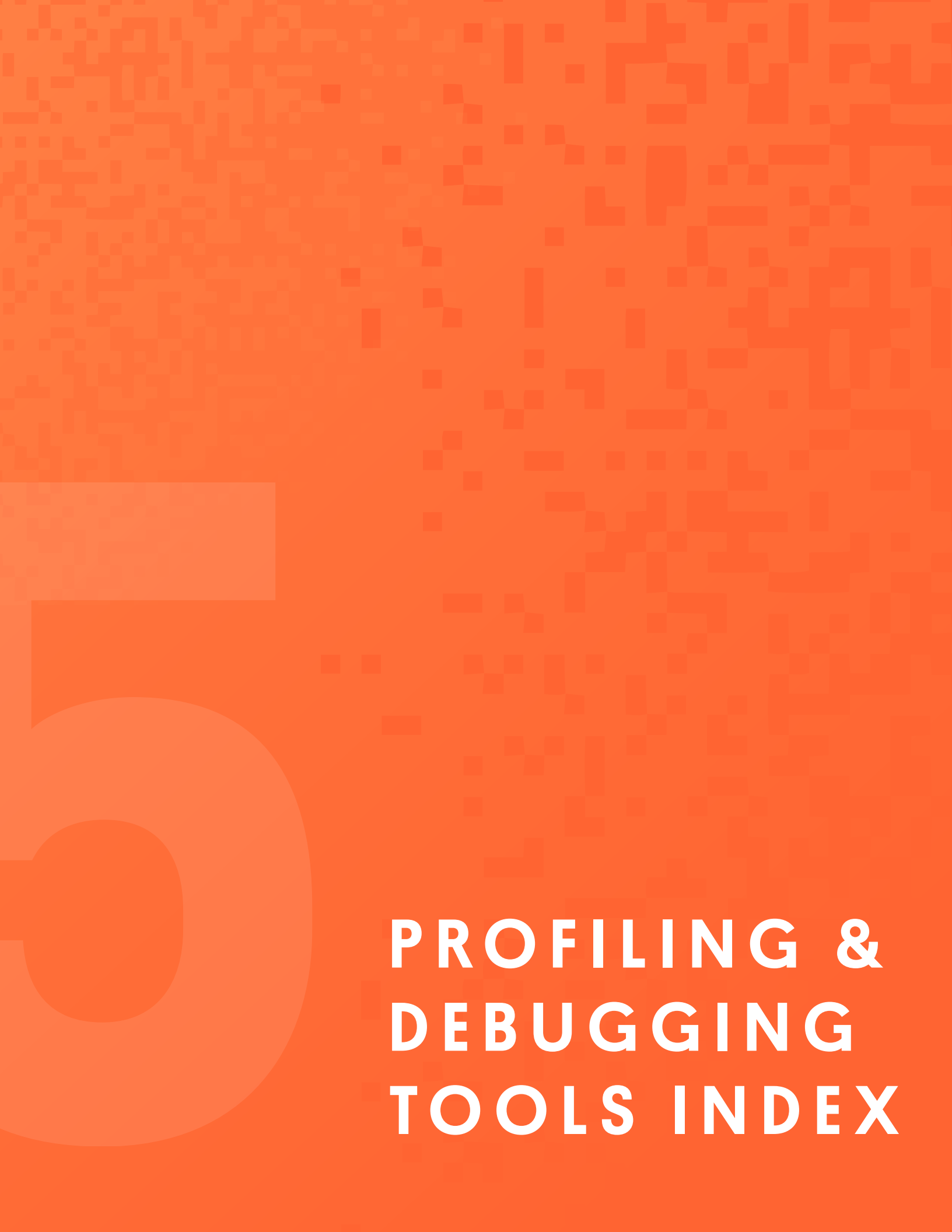
PROFILING &
DEBUGGING
TOOLS INDEX

Start your profiling with Unity's tools, and if you need greater detail, reach for the native profiliers 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 Mobile 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.

— 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.

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 Graphics Analyzer: Part of Arm's Mobile Studio software suite

— RenderDoc: GPU debugger for desktop and mobile platforms

— 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.

MORE RESOURCES

All Unity developers have access to a wealth of free resources to help them develop and deploy games and other real-time 3D and 2D interactive content. Find tips, tutorials, and inspiration on the Unity Blog, from the Resources collection for game developers, at Unity Learn, and on our Developer Tools page.

Unity®