



Optimize your game performance for consoles and PC

2020 LTS edition

 unity

Contents

Introduction	6
Profiling.....	7
Focus on optimizing the right areas	7
Understand how the Unity Profiler works.....	7
Deep Profiling	10
Use the Profile Analyzer.....	11
Work on a specific time budget per frame	11
Frames per second: A deceptive metric.....	12
Determine if you are GPU-bound or CPU-bound.....	13
Intel VTune.....	13
Project Auditor	13
Build Report Inspector	14
Memory.....	15
Use the Memory Profiler	15
Reduce the impact of garbage collection (GC)	16
Time garbage collection whenever possible	16
Use the Incremental Garbage Collector to split the GC workload	17
Heap Explorer	17
Programming and code architecture	18
Understand the Unity PlayerLoop	18
Minimize code that runs every frame	19
Cache the results of expensive functions.....	19
Avoid empty Unity events	20
Build a custom Update Manager.....	20

Remove Debug Log statements	21
Disable Stack Trace logging.....	22
Use hash values instead of string parameters	22
Choose the right data structure	22
Avoid adding components at runtime	22
Use object pools.....	23
Transform once, not twice.....	24
Use ScriptableObjects	24
Avoid lambda expressions	25
The C# Job System	25
The Burst compiler	27
Project configuration	28
Disable unnecessary Player or Quality settings.....	28
Switch to IL2CPP.....	28
Avoid large hierarchies.....	29
Assets	30
Compress textures	30
Texture import settings	31
Atlas your textures	32
Check your polygon counts	32
Mesh import settings	34
Other mesh optimizations	34
Audit your assets	35
The AssetPostprocessor	35
Asset Bundle Analyzer	35
Async texture buffer.....	35
Stream mipmaps and textures.....	36
Use Addressables.....	36

Graphics	38
Commit to a render pipeline.....	38
Render pipeline packages for consoles	39
Select a rendering path	40
Forward rendering path	40
Deferred shading path	41
Optimize Shader Graphs	42
Remove built-in shader settings	44
Strip shader variants	44
Particle simulations.....	45
Smooth with anti-aliasing.....	46
Common lighting optimizations.....	48
GPU optimization	52
Benchmark the GPU	52
Watch the rendering statistics	52
Use draw call batching.....	53
Check the Frame Debugger.....	55
Alternative debugging techniques.....	55
Optimize fill rate and reduce overdraw.....	56
Draw order and render queues	57
Optimizing graphics for consoles	59
Culling	64
Dynamic resolution.....	66
Multiple camera views	66
RenderObjects in URP	67
CustomPassVolumes in HDRP	67
Use Level of Detail (LOD)	68
Profile post-processing effects	69

User interface	70
Divide your Canvases.....	70
Hide invisible UI elements	70
Limit GraphicRaycasters and disable Raycast Target ..	70
Avoid layout groups	71
Avoid large List and Grid views	71
Avoid numerous overlaid elements	71
When using a fullscreen UI, hide everything else	72
Audio	73
Use lossless files as your source.....	73
Reduce your AudioClips	74
Optimize the AudioMixer	75
Physics	77
Simplify colliders	77
Optimize your settings	78
Adjust simulation frequency.....	79
Modify CookingOptions for MeshColliders.....	81
Use Physics.BakeMesh.....	81
Use Box Pruning for large scenes	82
Modify solver iterations	83
Disable automatic transform syncing	83
Reuse Collision Callbacks.....	84
Move static colliders.....	84
Use non-allocating queries	85
Batch queries for ray casting	85
Visualize with the Physics Debugger	85

Animation	86
Use generic rather than humanoid rigs.	87
Use alternatives for simple animation.	87
Avoid scale curves	88
Update only when visible	88
Optimize workflow	88
Workflow and collaboration	89
Use version control	89
Plastic SCM	89
Break up large Scenes	90
Speed up sharing with Unity Accelerator	91
Access the help you need with Unity Integrated Success	91
Get a Project Review	92
Developer Relations Manager (DRM)	92
Next steps	93

Introduction

Optimization will continue to be one of your challenges as a developer.

Getting your game to run with fewer resources and at higher frame rates ensures that it can reach more potential players – and keep them engaged.

While your audience may take it for granted that your game runs at silky-smooth 60+ frames per second (fps), achieving your performance goals across multiple platforms is not always easy. It requires effort to make both your code architecture and art assets more efficient.

This guide assembles knowledge and advice from Unity's expert software engineers. Our Accelerate Solutions games team has tested these best practices with our industry partners in real-world scenarios. We're here to help you identify key areas for optimization in your Unity project.

Profiling

The [Unity Profiler](#) provides performance information about your application – but it can't help you if you don't use it.

Profile your project early in development, not just when you are close to shipping. Investigate glitches or spikes as soon as they appear. As you develop a “performance signature” for your project, you'll be able to spot new issues more easily.

While profiling in the Editor can give you an idea of the relative performance of different systems in your game, profiling on each device gives you the opportunity to gain more accurate insights. Profile a development build on target devices whenever possible. Remember to profile and optimize for both the highest- and lowest-spec devices that you plan to support.

See [Profiling Applications Made with Unity](#) for more information.

Focus on optimizing the right areas

Don't guess or make assumptions about what is slowing down your game's performance. Use the Unity Profiler and platform-specific tools to locate the precise source of a lag.

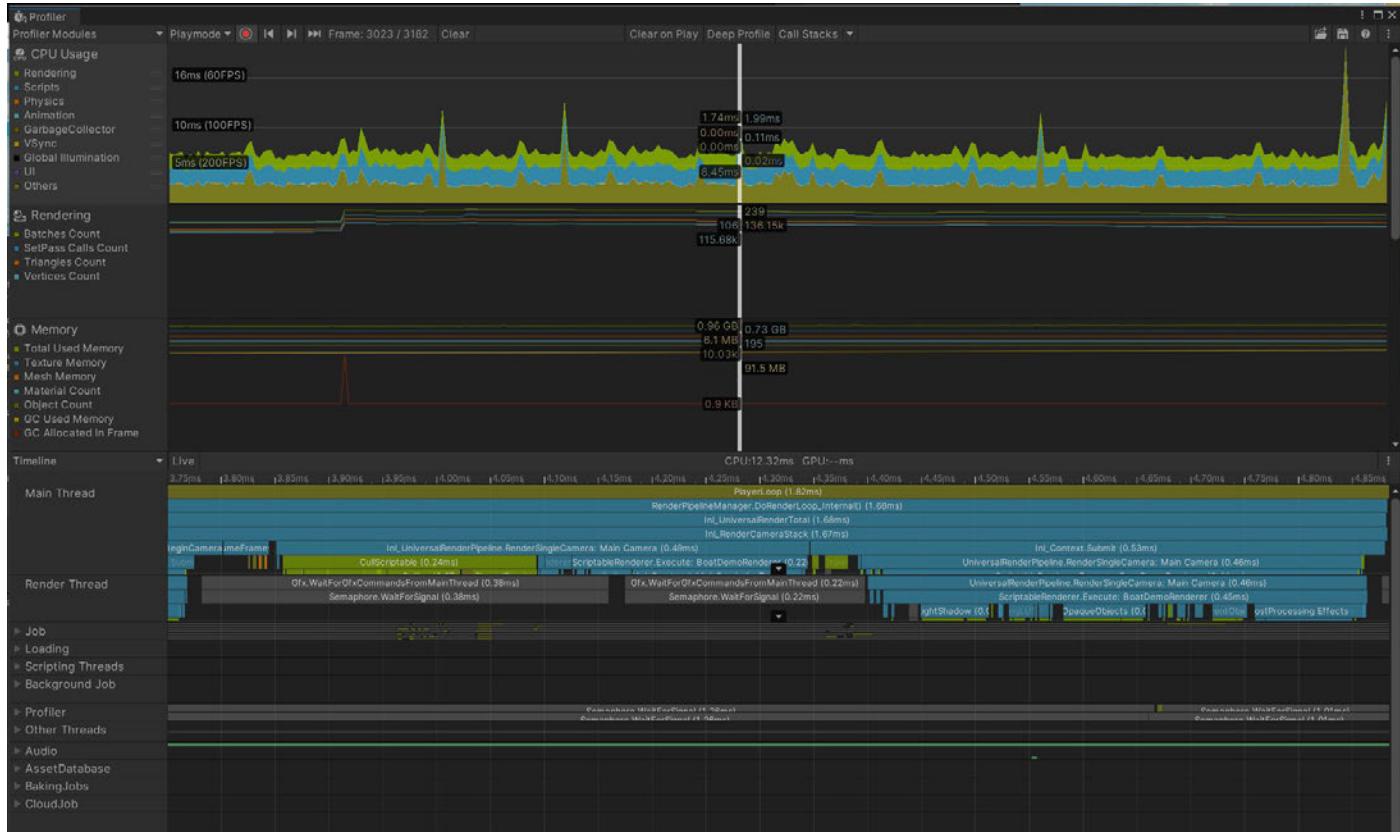
Of course, not every optimization described here will apply to your application. Something that works well in one project may not translate to yours. Identify genuine bottlenecks and concentrate your efforts on what benefits your work.

Understand how the Unity Profiler works

The Unity Profiler can help 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.

It's an instrumentation-based profiler that profiles code timings explicitly wrapped in [ProfilerMarkers](#) (such as MonoBehaviour's Start or Update methods, or specific API calls).

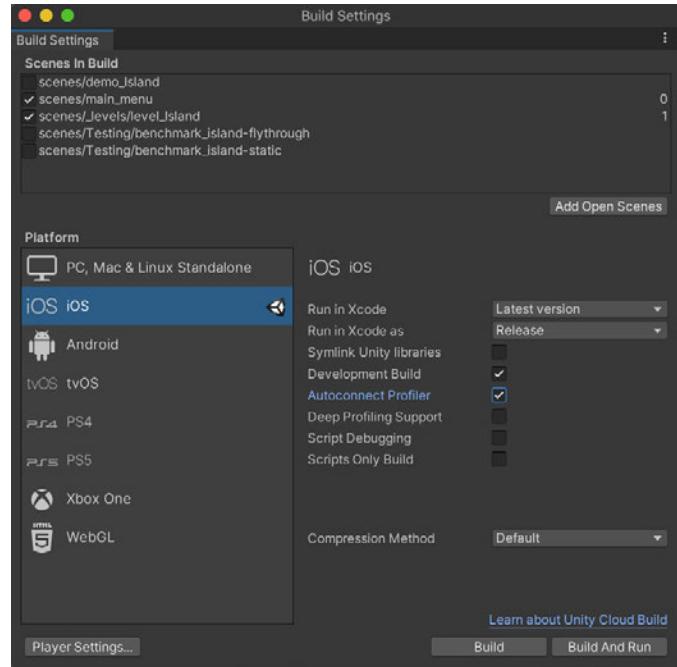
Begin by enabling the CPU and Memory tracks as your default. You can monitor supplementary Profiler Modules like Renderer, Audio, and Physics as needed for your game (for example, profiling for physics-heavy or music-based gameplay). However, only enable what you need so you don't impact performance and skew your results.



Use the Unity Profiler to test performance and resource allocation.

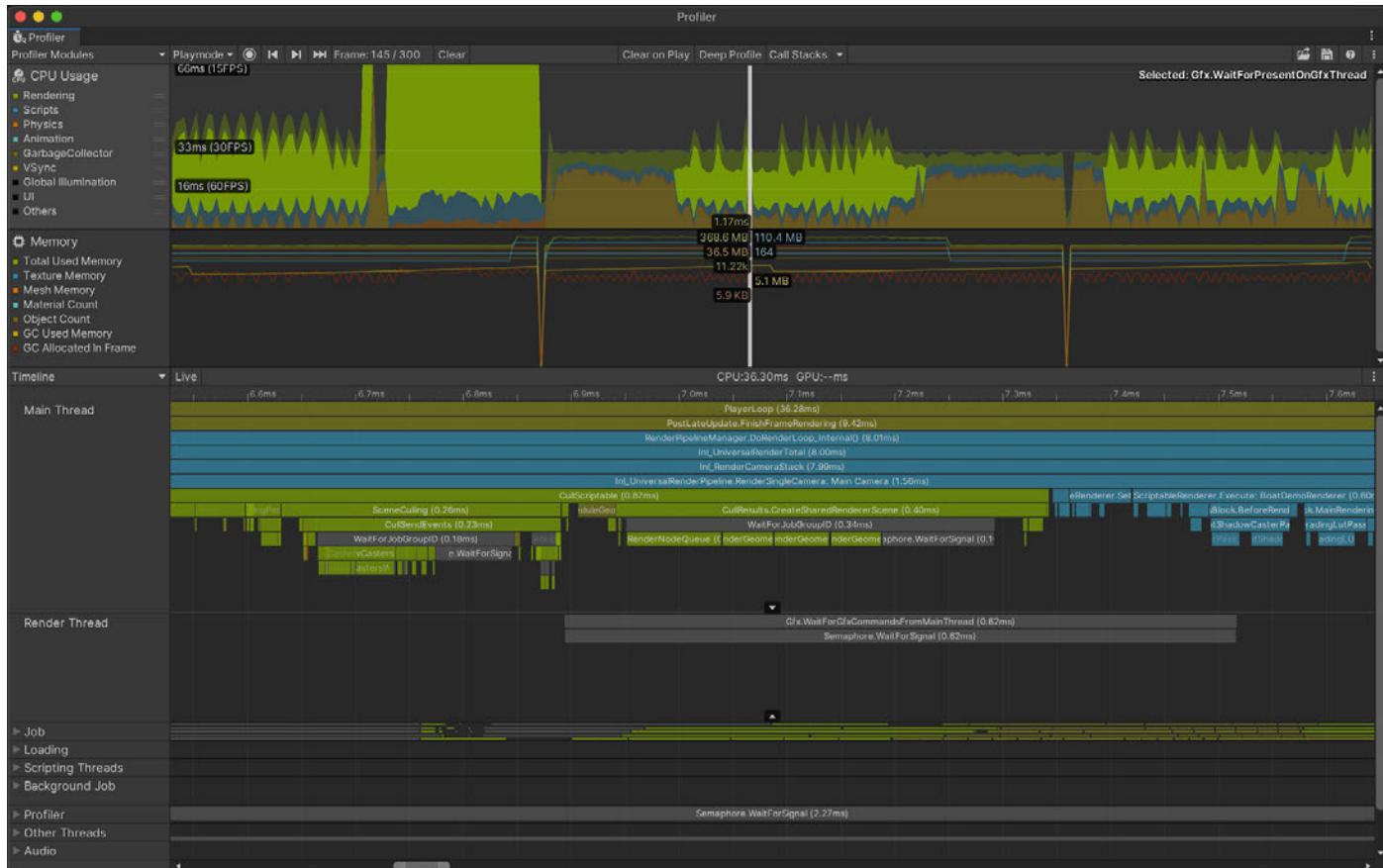
To capture profiling data from an actual device within your chosen platform, check the **Development Build** before you click Build and Run. Then, connect the Profiler to your application manually once it's running.

You can optionally check **Autoconnect Profiler** in the Build Options. Sometimes this is useful if you specifically want to capture the first few frames of the application. Be warned that this option can add 5–10 seconds of startup time, so only use it when necessary.



Adjust your Build Settings before profiling.

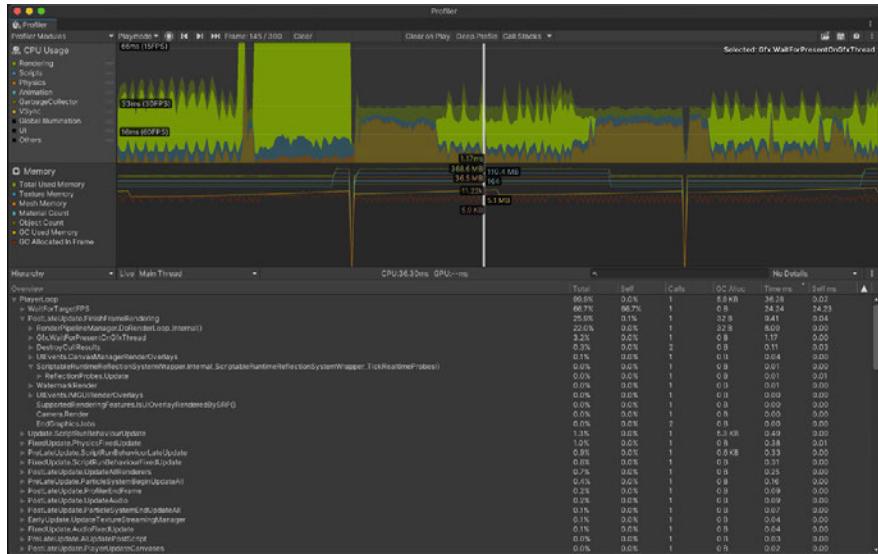
Choose the platform target to profile. The Record button tracks several seconds of your application's playback (300 frames by default). Go to **Unity > Preferences > Analysis > Profiler > Frame Count** to increase this up to 2000 if you need longer captures. While this costs more CPU and memory resources, it can be useful depending on your specific scenario.



Use the Timeline view to determine if you are CPU-bound or GPU-bound.

Click in the window to analyze a specific frame. Next, use either the Timeline or Hierarchy view for the following:

- **Timeline** shows the visual breakdown of timing for a specific frame. This allows you to visualize how the activities relate to one another and across different threads. Use this option to determine if you are CPU- or GPU-bound.
- **Hierarchy** shows the hierarchy of ProfileMarkers, grouped together. This allows you to sort the samples based on time cost in milliseconds (Time ms and Self ms). You can also count the number of Calls to a function and the managed heap memory (GC Alloc) on the frame.



The Hierarchy view allows you to sort ProfileMarkers by time cost.

Read a complete overview of the Unity Profiler [here](#). Those new to profiling can also watch this Unite Now session, [Introduction to Unity Profiling](#).

Before optimizing anything in your project, save the Profiler .data file. Implement your changes and compare the saved .data before and after the modification. Rely on this cycle to improve performance: profile, optimize, and compare.

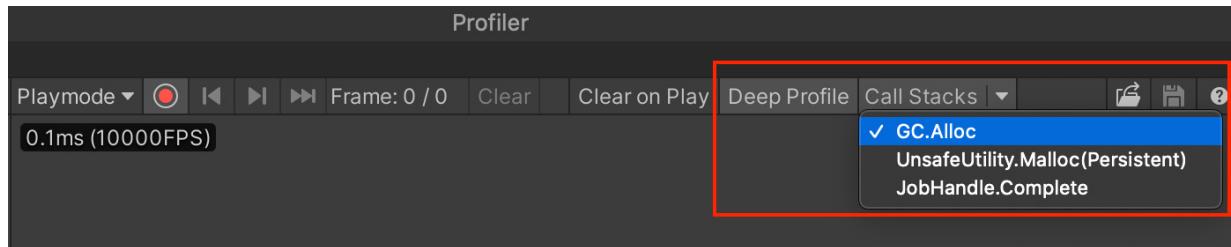
Deep Profiling

You can also enable [Deep Profiling Support](#) in the Build Settings. When the built Player starts, the Deep Profiler profiles every part of your code, not just code timings explicitly wrapped in [ProfilerMarkers](#).

When Deep Profiling is enabled, Unity can profile the beginning and end of every function call in your script code. This can help you identify exactly which part of your application is causing a slowdown.

However, Deep Profiling is resource-intensive and uses a lot of memory. Each ProfilerMarker adds a tiny bit of overhead (about 10ns, depending on the platform), so each additional point of measurement slows your application more. Also, be warned that if you have a lot of function calls, deep profiling amplifies their overhead as well.

If you want to see more details on samples with markers such as [GC.Alloc](#) or [JobHandle.Complete](#), navigate to the Profiler window toolbar and enable the **Call Stacks** setting. This provides the sample's full call stack, which gives you the information you need without incurring the overhead of Deep Profiling.



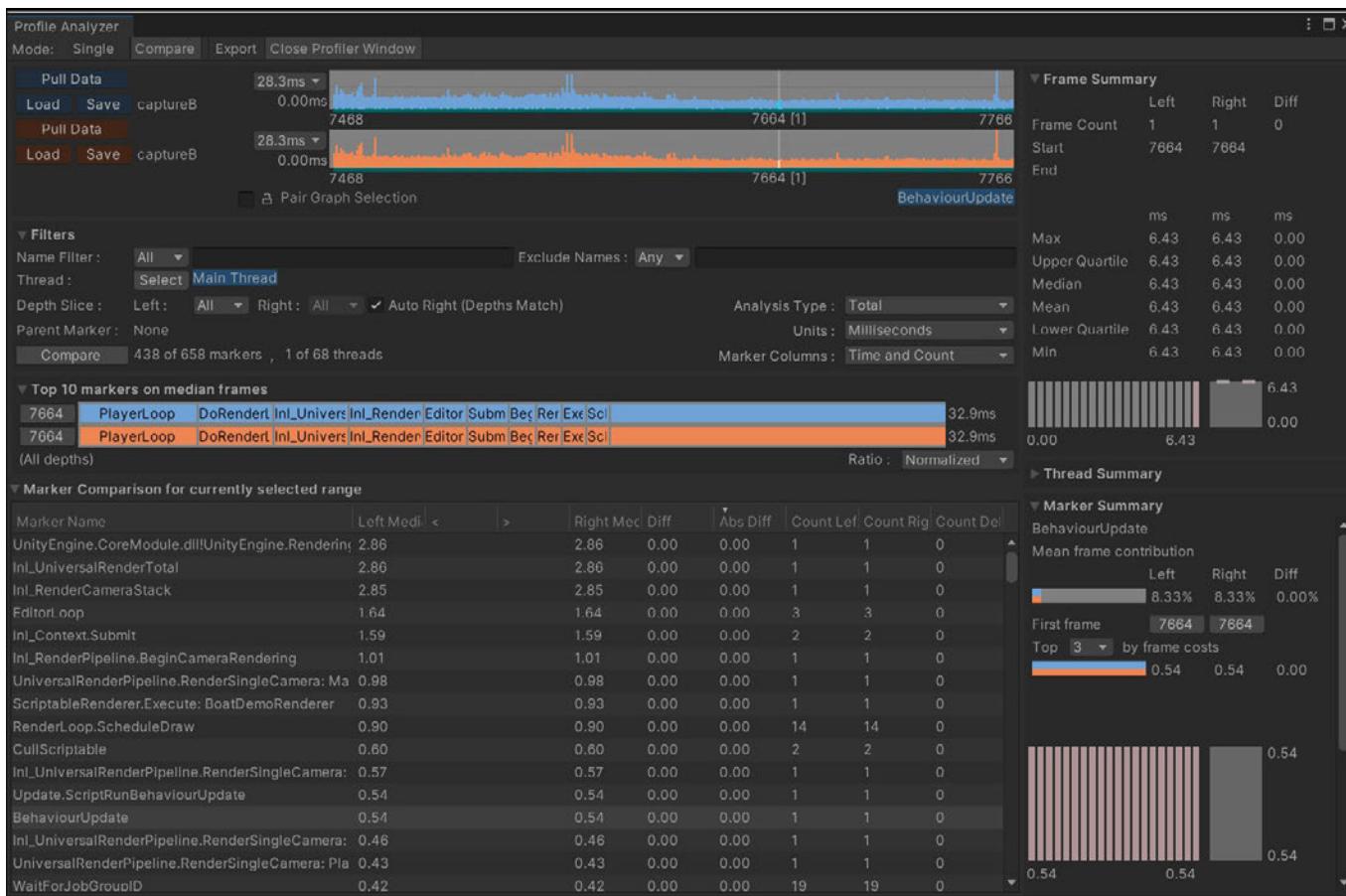
Use Call Stacks instead of Deep Profiling whenever possible.

In general, only use Deep Profiling when it's necessary, since your application runs significantly slower when it's in use.

Use the Profile Analyzer

This tool lets you aggregate multiple frames of Profiler data, then locate frames of interest. Want to see what happens to the Profiler after you make a change to your project? The Compare view lets you load and differentiate two data sets, so you can test changes and see whether they improve performance.

The [Profile Analyzer](#) is available via Unity's Package Manager.



Take an even deeper dive into frames and marker data with the [Profile Analyzer](#), which complements the existing Profiler.

Work on a specific time budget per frame

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.

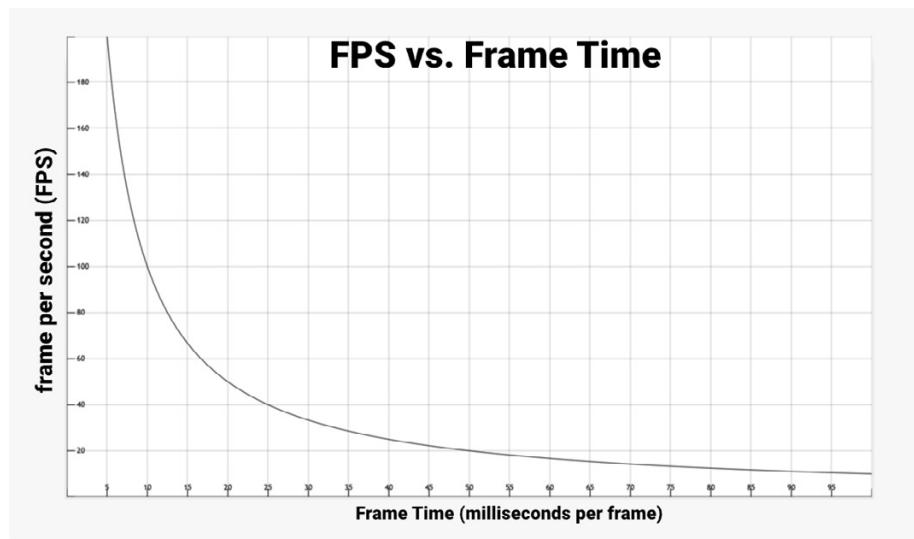
You can exceed this budget during non-interactive sequences (e.g., for cutscenes or loading scenes), but not during actual gameplay. Even a single frame that exceeds the target frame budget will cause frame rate hitches.

Frames per second: A deceptive metric

A common way that gamers measure performance is with frame rate, or frames per second. This can be a deceptive metric when gauging your application's performance.

We recommend that you use frame time in milliseconds instead.

To understand why, look at this graph of **fps versus Frame Time**:



fps versus Frame Time

Consider these numbers:

$$1000 \text{ ms/sec} / 900 \text{ fps} = 1.111 \text{ ms per frame}$$

$$1000 \text{ ms/sec} / 450 \text{ fps} = 2.222 \text{ ms per frame}$$

$$1000 \text{ ms/sec} / 60 \text{ fps} = 16.666 \text{ ms per frame}$$

$$1000 \text{ ms/sec} / 56.25 \text{ fps} = 17.777 \text{ ms per frame}$$

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.

Determine if you are GPU-bound or CPU-bound

The Profiler can tell you if your CPU is taking longer than your allotted frame budget, or if the culprit is your GPU. It does this by [emitting markers prefixed with Gfx](#) as follows:

- If you see the **Gfx.WaitForCommands** marker, it means that the render thread is ready, but you might be waiting for a bottleneck on the main thread.
- If you frequently encounter **Gfx.WaitForPresentOnGfxThread**, it means that the main thread was ready but was waiting for the render thread. This might indicate that your application is GPU-bound. Check the [CPU Profiler module's Timeline view](#) to see activity on the render thread.

If the render thread spends time in **Camera.Render**, your application is CPU-bound and might be spending too much time sending draw calls or textures to the GPU.

If the render thread spends time in **Gfx.PresentFrame**, your application is GPU-bound or might be [waiting for VSync](#) on the GPU.

Refer to the [Common Profiler markers](#) documentation for a complete list of markers. Also, check out our blog post on [Fixing Time.deltaTime in Unity 2020.2 for smoother gameplay](#) for more information about the frame pipeline.

Intel VTune

The [Intel VTune Profiler](#) assesses bottlenecks in your CPU, GPU, and I/O with standalone builds on Intel-based hardware. Use it to isolate slow frames and tasks and make sense of what's going on with your application.

Download the VTune Profiler, then configure Unity before making a standalone build:

- Set the **Scripting Backend** to **IL2CPP** in the Player Settings
- Set **C++ Compiler Configuration** to **Release** in the Player Settings
- Enable **Copy PDB files** and **Development Build** in the Build Settings

Then attach the standalone build to the Intel Profiler. VTune can go beyond the Unity Profiler and profile the application start time or which functions are taking the most resources over a given period of time.

Project Auditor

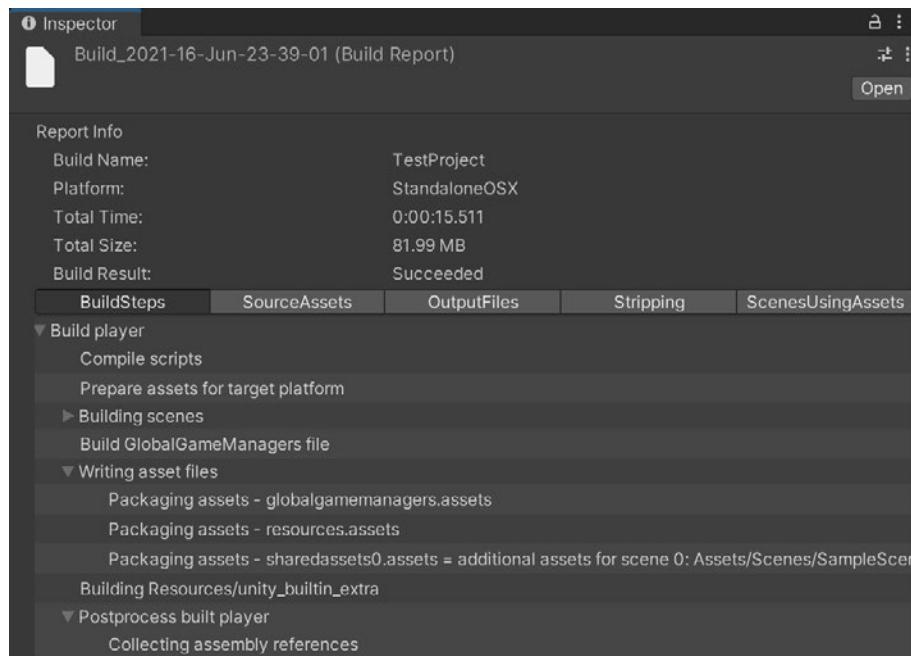
The [Project Auditor](#) is an experimental tool capable of performing static analysis of a project's scripts and settings. It offers a great way to track down the causes of managed memory allocations, inefficient project configurations, and possible performance bottlenecks.

The Project Auditor is a free, unofficial package for use with the Editor. For information, please refer to the [Project Auditor documentation](#).

Build Report Inspector

The [Build Report Inspector](#) (in Preview) is an Editor script that lets you access information about your last build so you can profile the time spent building your project and the build's disk size footprint.

This script allows you to inspect this information graphically in the Editor UI, making it more easily accessible than the script APIs would.



The Build Report Inspector

The build report displays statistics on included resources and generated code size.

Watch a Unite Now presentation on [Optimizing Binary Deployment Size](#) to learn how to optimize your build size. You can also read the [Build Report Inspector documentation](#) for more information.

PIX (Xbox/Windows)

PIX is a performance tuning and debugging tool for game developers. It is available for both [Xbox](#) (as part of the [GDKX](#)) and [Windows](#).

PIX offers multiple modes of analysis:

- **GPU analysis** for profiling and debugging graphics rendering. Hardware-specific counters are exposed on both Xbox and Windows.
- **Timing analysis** for understanding the performance of all CPU and GPU work, file IO, and memory allocations over extended periods of time, including a powerful sampling profiler and metrics analyzer.

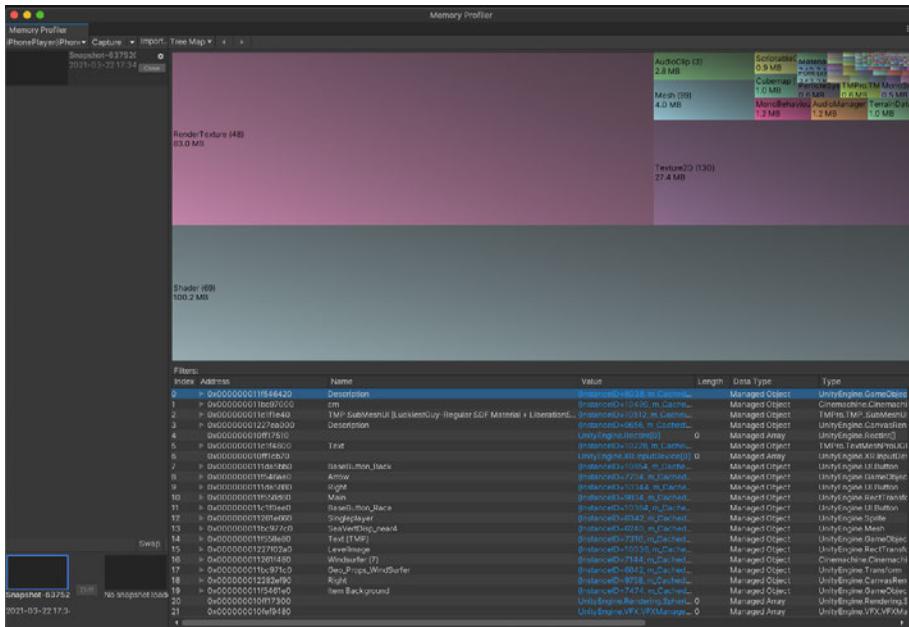
Memory

Unity employs automatic memory management for your user-generated code and scripts. Small pieces of data, like value-typed local variables, are allocated to the stack. Larger pieces of data and longer-term storage are allocated to the managed heap.

The garbage collector periodically identifies and deallocates unused heap memory. While this runs automatically, the process of examining all the objects in the heap can cause the game to stutter or run slowly.

Optimizing your memory usage means being conscious of when you allocate and deallocate managed heap memory, and how you minimize the effect of garbage collection.

See [Understanding the managed heap](#) for more information.



Capture, inspect, and compare snapshots in the Memory Profiler.

Use the Memory Profiler

This separate add-on (available as a Preview package in the Package Manager) can take a snapshot of your memory to help you identify problems like fragmentation and memory leaks.

Click in the Tree Map view to trace a variable to the native object holding onto memory. Here, you can identify common memory consumption issues, like excessively large textures or duplicate assets.

Learn how to leverage the [Memory Profiler in Unity](#) for improved memory usage.

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 certain unnecessary heap allocations, which could 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**.

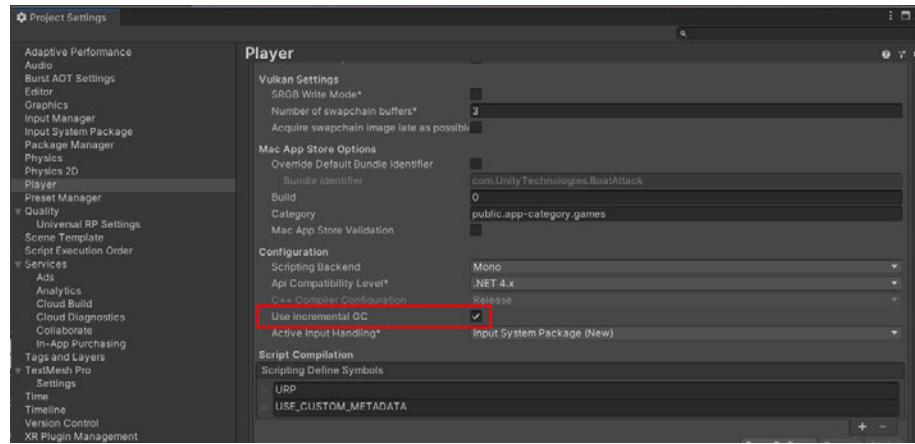
See [Understanding Automatic Memory Management](#) for examples of how to use this to your advantage.¹

¹Note that using the GC can add read-write barriers to some C# calls, which come with little overhead that can add up to ~1 ms per frame of scripting call overhead. For optimal performance, it is ideal to have no GC Allocs in the main gameplay loops and to hide the GC.Collect where a user won't notice it.

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, much shorter interruptions that distribute the workload over many frames. If garbage collection is impacting performance, try enabling 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: Incremental GC can temporarily help mitigate garbage collection issues, but the best long-term course of action is to locate and stop frequent allocations that trigger garbage collection.

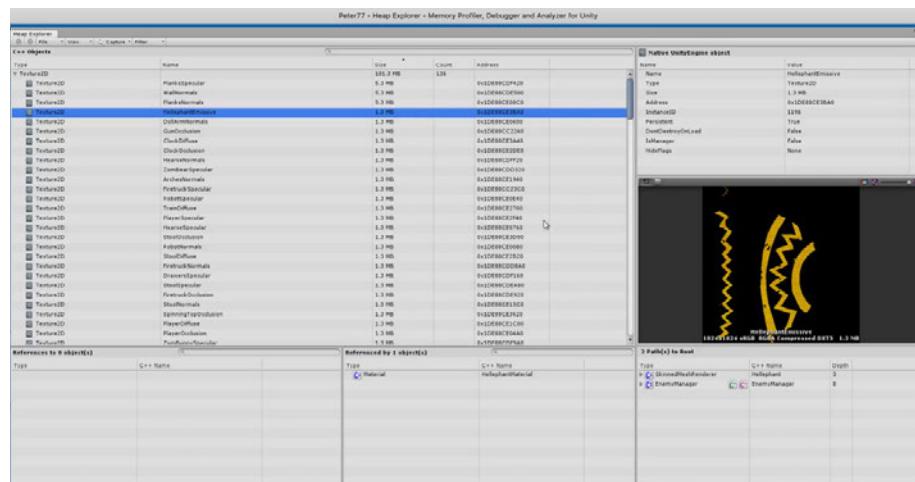


Use the Incremental Garbage Collector to reduce GC spikes.

Heap Explorer

[Heap Explorer](#) is a third-party Memory Profiler, Debugger, and Analyzer for Unity. The package can be used to grab a memory snapshot of a given frame and shows clear tables for Native, Managed, and Static memory. Heap Explorer can help you identify duplicated assets, like textures copied between multiple AssetBundles.

Though it overlaps in functionality with Unity's Memory Profiler, some still prefer Heap Explorer for its easy to understand UI/UX.

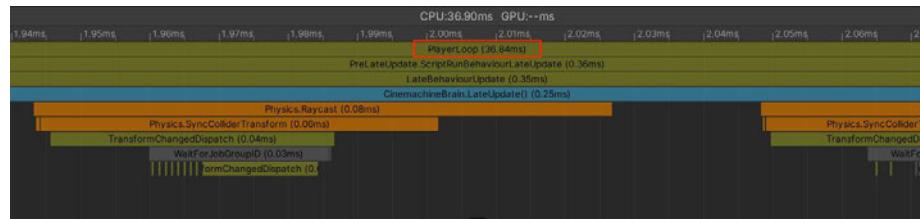


The Heap Explorer

Programming and code architecture

The Unity [PlayerLoop](#) contains functions for interacting with the core of the game engine. This structure includes a number of systems that handle initialization and per-frame updates. All of your scripts will rely on this PlayerLoop to create gameplay.

When profiling, you'll see your project's user code under the PlayerLoop (with Editor components under the EditorLoop).



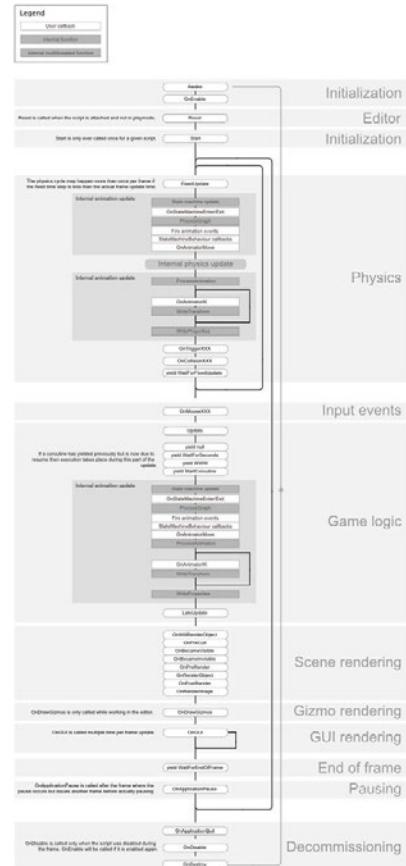
The Profiler will show your custom scripts, settings, and graphics in the context of the entire engine's execution.

Understand the Unity PlayerLoop

Make sure you understand the [execution order](#) of Unity's frame loop. Every Unity script runs several event functions in a predetermined order. You should understand the difference between Awake, Start, Update, and other functions that create the lifecycle of a script.

Refer to the [Script Lifecycle Flowchart](#) for event functions' specific order of execution.

Get to know the PlayerLoop and the lifecycle of a script.



Minimize code that runs every frame

Consider whether code must run every frame. Move unnecessary logic out of Update, LateUpdate, and FixedUpdate. These event functions are convenient places to put code that must update every frame, while extracting any logic that does not need to update with that frequency. Whenever possible, only execute logic when things change.

If you *do* need to use Update, consider running the code every n frames. This is one way to apply time slicing, a common technique of distributing a heavy workload across multiple frames. In this example, we run the ExampleExpensiveFunction once every three frames:

```
private int interval = 3;

void Update()
{
    if (Time.frameCount % interval == 0)
    {
        ExampleExpensiveFunction();
    }
}
```

The trick is to interleave this with other work that runs on the other frames. In this example, you could “schedule” other expensive functions when **Time.frameCount % interval == 1** or **Time.frameCount % interval == 2**.

Alternatively, use a custom UpdateManager class (below) and update subscribed objects every n frames.

Cache the results of expensive functions

GameObject.Find, GameObject.GetComponent, and Camera.main (in versions [prior to 2020.2](#)) can be expensive, so it’s best to avoid calling them in Update methods. Also, avoid placing expensive methods in OnEnable and OnDisable if they are called often.

Frequently calling these methods can contribute to CPU spikes. Wherever possible, run expensive functions in the initialization phase (i.e., [MonoBehaviour.Awake](#) and [MonoBehaviour.Start](#)). Cache the needed references and reuse them later.

Here’s an example that demonstrates inefficient use of a repeated GetComponent call:

```
void Update()
{
    Renderer myRenderer = GetComponent<Renderer>();
    ExampleFunction(myRenderer);
}
```

Instead, invoke **GetComponent** only once, as the result of the function is cached. The cached result can be reused in **Update** without any further calls to **GetComponent**.

```
private Renderer myRenderer;

void Start()
{
    myRenderer = GetComponent<Renderer>()
}

void Update()
{
    ExampleFunction(myRenderer);
}
```

Avoid empty Unity events

Even empty MonoBehaviours require resources, so you should remove blank Update or LateUpdate methods.

Use preprocessor directives if you are employing these methods for testing:

```
#if UNITY_EDITOR
void Update()
{
}
#endif
```

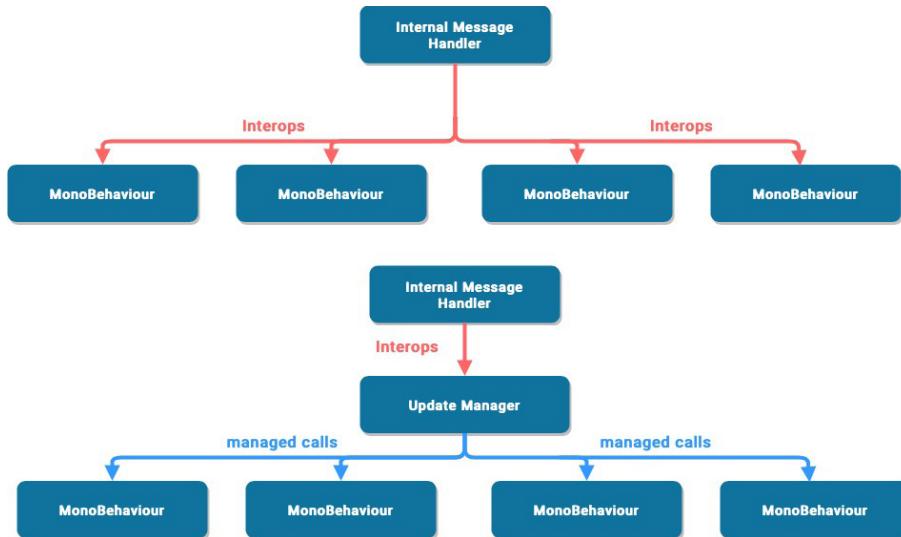
Here, you can freely use the Update in-Editor for testing without unnecessary overhead slipping into your build. This blog post on [10,000 Update calls](#) will help you understand how Unity executes [Monobehaviour.Update](#).

Build a custom Update Manager

A common usage pattern for Update or LateUpdate is to run logic only when some condition is met. This can lead to a lot of per-frame callbacks that effectively run no code except for checking this condition.

Every time Unity calls a Message method like Update or LateUpdate, it makes an *interop* call, a call from the C/C++ side to the managed C# side. For a small number of objects, this is not an issue. When you have thousands of objects, this overhead starts becoming significant.

Consider creating a custom UpdateManager if you have a large project using Update or LateUpdate in this fashion (e.g., an open-world game). Have active objects subscribe to this UpdateManager when they want callbacks, and unsubscribe when they don't. This pattern could reduce many of the interop calls to your MonoBehaviour objects.



Building a custom Update Manager reduces interop calls.

Refer to [Game engine-specific optimization techniques for Unity](#) to see an example of implementation and potential performance gains.

Remove Debug Log statements

Log statements (especially in Update, LateUpdate, or FixedUpdate) can bog down performance. Disable your Log statements before making a build.

To do this more easily, consider making a [Conditional attribute](#) along with a preprocessing directive. For example, create a custom class like this:

```

public static class Logging
{
    [System.Diagnostics.Conditional("ENABLE_LOG")]
    static public void Log(object message)
    {
        UnityEngine.Debug.Log(message);
    }
}

```



Adding a custom preprocessor directive lets you partition your scripts.

Generate your log message with your custom class. If you disable the **ENABLE_LOG** preprocessor in the **Player Settings > Scripting Define Symbols**, all of your Log statements disappear in one fell swoop.

Handling strings and text is a common source of performance problems in Unity projects. Removing Log statements – and their expensive string formatting – can be a huge win.

Disable Stack Trace logging

Use the Stack Trace options in the Player Settings to control what type of log messages appear.

If your application is logging errors or warning messages in your release build (e.g., to generate crash reports in the wild), disable stack traces to improve performance.

Stack Trace*	None	ScriptOnly	Full
Error	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Assert	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Warning	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Log	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Exception	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Stack Trace options

Use hash values instead of string parameters

Unity does not use string names to address Animator, Material, and Shader properties internally. For speed, all property names are hashed into property IDs, and these IDs are actually used to address the properties.

When using a Set or Get method on an Animator, Material, or Shader, harness the integer-valued method instead of the string-valued methods. The string methods simply perform string hashing and then forward the hashed ID to the integer-valued methods.

Use [Animator.StringToHash](#) for Animator property names and [Shader.PropertyToID](#) for Material and Shader property names.

Choose the right data structure

Your choice of data structure impacts efficiency as you iterate thousands of times per frame. Not sure whether to use a List, Array, or Dictionary for your collection? Follow the [MSDN guide to data structures](#) in C# as a general guide for choosing the correct structure.

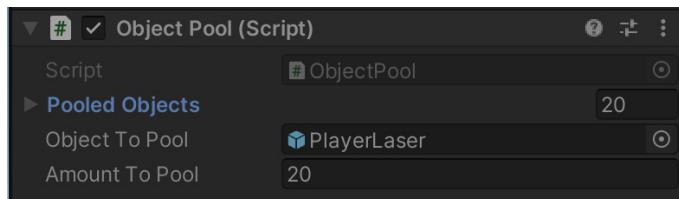
Avoid adding components at runtime

Invoking AddComponent at runtime comes with some cost. Unity must check for duplicates or other required components whenever adding components at runtime.

[Instantiating a Prefab](#) with the desired components already set up is generally more performant.

Use object pools

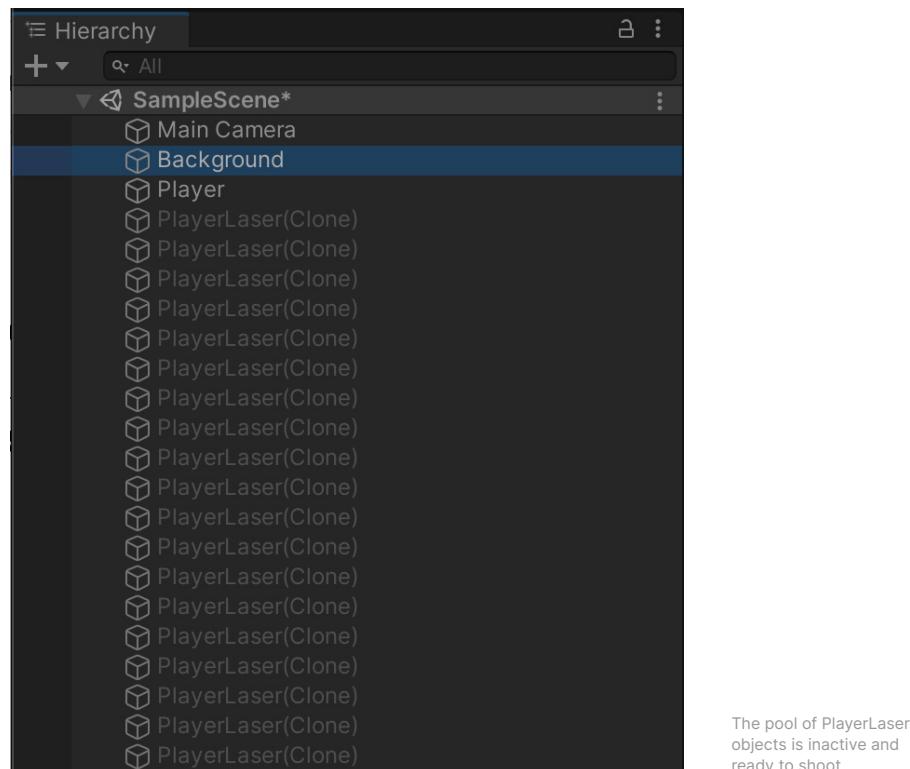
Instantiate and Destroy can generate garbage and garbage collection (GC) spikes, and this is generally a slow process. Rather than regularly instantiating and destroying GameObjects (e.g., shooting bullets from a gun), use [pools](#) of preallocated objects that can be reused and recycled.



Create the reusable instances at a point in the game (e.g., during a menu screen or a loading screen) when a CPU spike is less noticeable. Track this “pool” of objects with a collection. During gameplay, simply enable the next available instance when needed, disable objects instead of destroying them, and return them to the pool.

This reduces the number of managed allocations in your project and can prevent garbage collection problems.

Learn how to create a simple Object Pooling system in Unity [here](#).



Transform once, not twice

When moving Transforms, use [Transform.SetPositionAndRotation](#) to update both position and rotation at once. This avoids the overhead of modifying a transform twice.

If you need to [Instantiate](#) a GameObject at runtime, a simple optimization is to parent and reposition during instantiation:

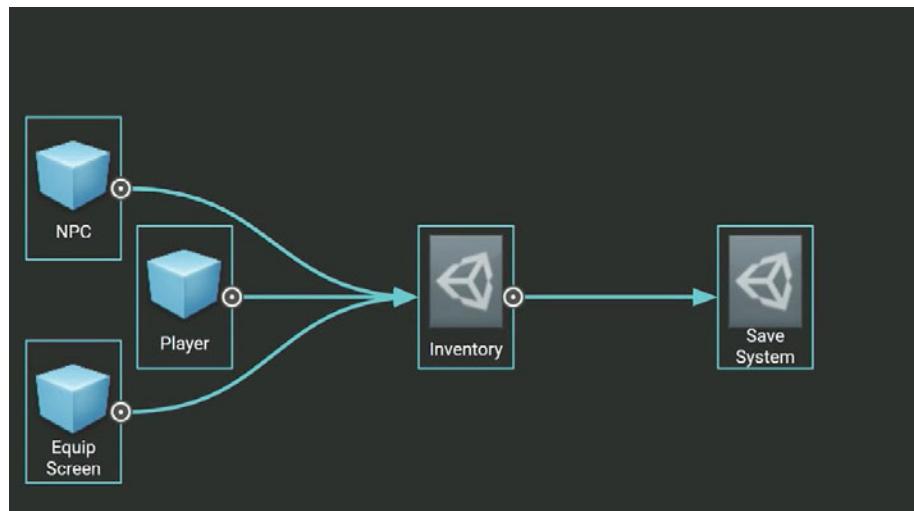
```
GameObject.Instantiate(prefab, parent);
GameObject.Instantiate(prefab, parent, position, rotation);
```

For more on Object.Instantiate, please see the [Scripting API](#).

Use ScriptableObjects

Store unchanging values or settings in a ScriptableObject instead of a MonoBehaviour. The ScriptableObject is an asset that lives inside of the project that you only need to set up once. It cannot be directly attached to a GameObject.

Create fields in the ScriptableObject to store your values or settings, then reference the ScriptableObject in your MonoBehaviours.



In this example, a ScriptableObject called Inventory holds settings for various GameObjects.

Using fields from the ScriptableObject can prevent unnecessary duplication of data every time you instantiate an object with that MonoBehaviour.

Watch this [Introduction to ScriptableObjects](#) tutorial to see how ScriptableObjects can help your project. You can also find relevant documentation [here](#).

Avoid lambda expressions

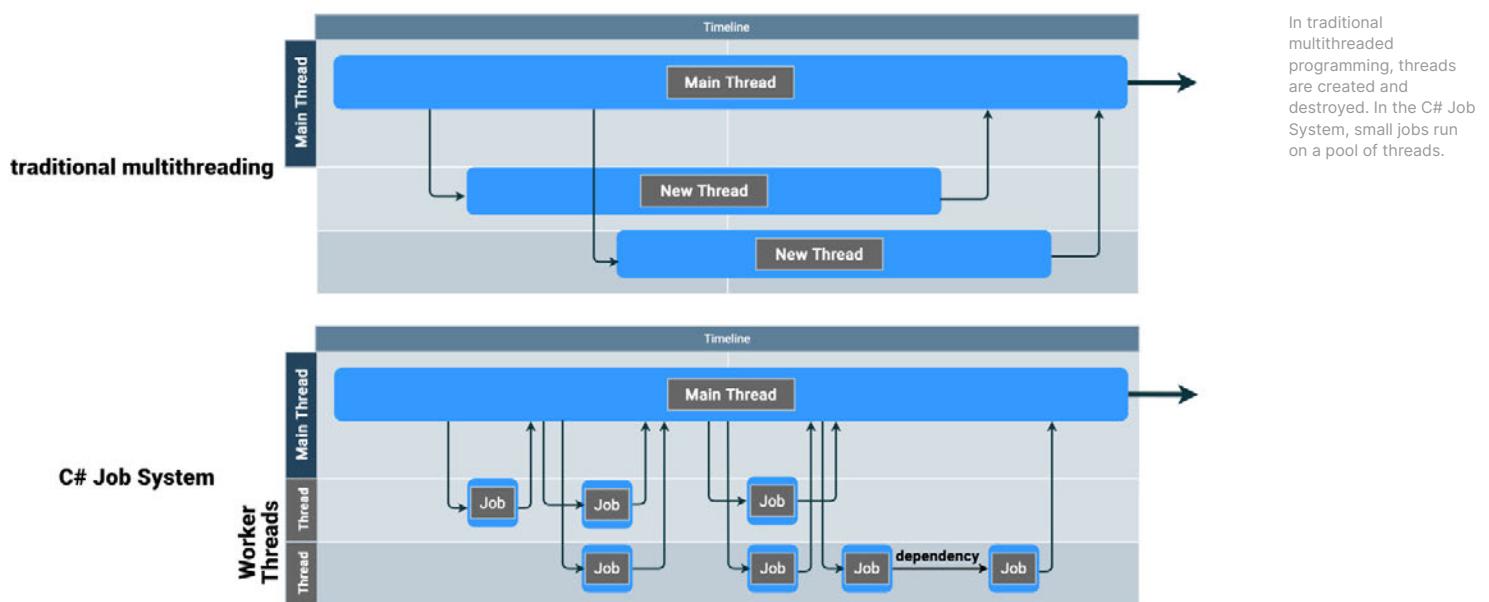
A lambda expression can simplify your code, but that simplification comes at a cost. Calling a lambda creates a delegate as well. Passing context (e.g., `this`, an instance member, or a local variable) into the lambda invalidates any caching for the delegate. When that happens, invoking it frequently can generate significant memory traffic.

Refactor any methods containing closures while using lambda expressions. See an example of how to do that [here](#).

The C# Job System

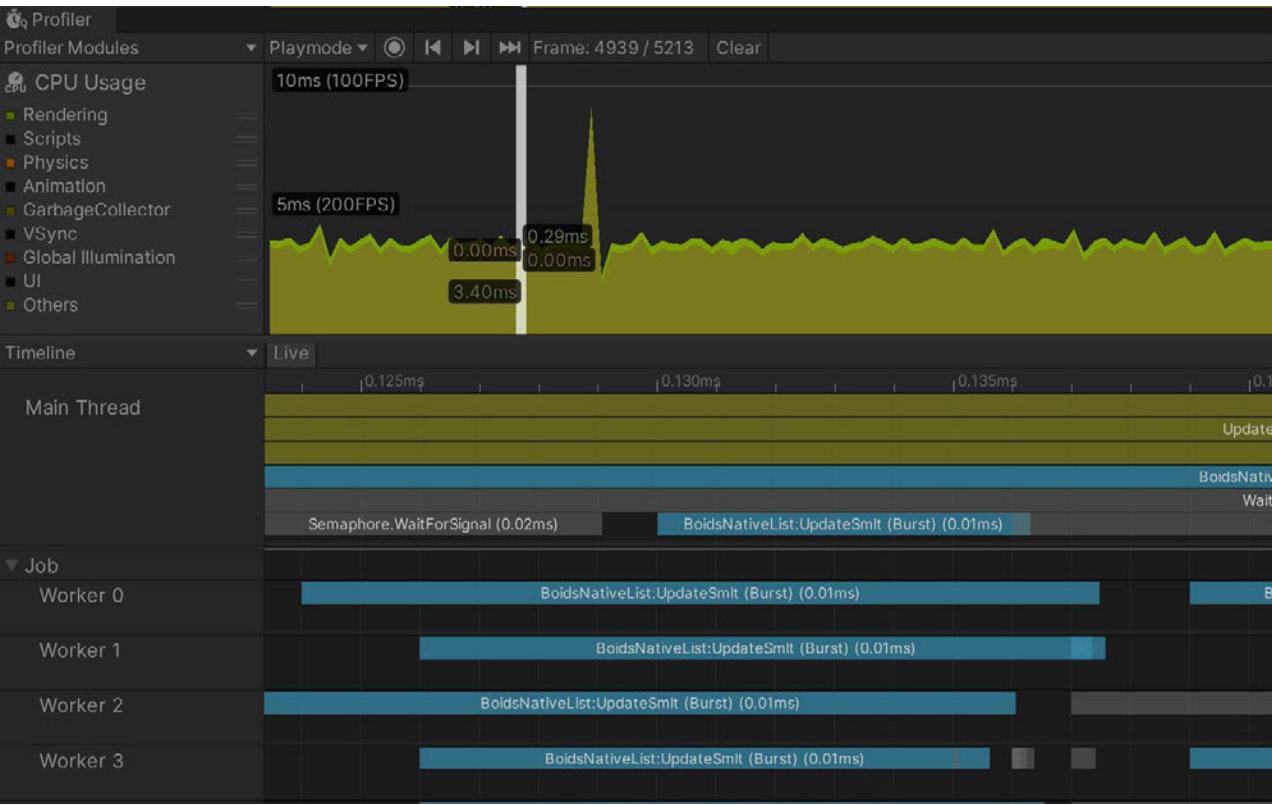
Modern CPUs have multiple cores, but your application needs multithreaded code to take advantage of them. Unity's Job System allows you to split large tasks into smaller chunks that run in parallel on those extra CPU cores, which can improve performance significantly.

Often in multithreaded programming, one CPU thread of execution, the *main thread*, creates other threads to handle tasks. These additional *worker threads* then synchronize with the main thread once their work completes.



If you have a few tasks that run for a long time, this approach to multithreading works well. However, it's less efficient for a game application, which must typically process many short tasks at 30–60 frames per second.

That's why Unity uses a slightly different approach to multithreading called the [C# Job System](#). Rather than generate many threads with a short lifetime, it breaks your work into smaller units called *jobs*.



Timeline view in the Profiler shows jobs running on the worker threads.

These jobs go into a [queue](#), which schedules them to run on a shared pool of [worker threads](#). [JobHandles](#) help you create dependencies, ensuring that the jobs run in the correct order.

One potential issue with multithreading is a [race condition](#), which occurs when two threads access a shared variable at the same time. To prevent this, Unity multithreading uses a safety system to isolate the data a job needs to execute. The C# Job System launches each job with a *copy* of the job structure, eliminating race conditions.

To use Unity's C# Job System, follow these guidelines:

- Change classes to be structs. A job is any struct that implements the [IJob](#) interface. If you're performing the same task on a large number of objects, you could also use [IJobParallelFor](#) to run across multiple cores.
- Data passed into a job must be [blittable](#). Remove reference types and pass only the blittable data into the job as a copy.
- Because the work within each job remains isolated for safety, you send the results back to the main thread using a [NativeContainer](#). A NativeContainer from the [Unity Collections package](#) provides a C# wrapper for native memory. Its subtypes (e.g., `NativeArray`, `NativeList`, `NativeHashMap`, `NativeQueue`, etc.)² work like their equivalent C# data structures.

Refer to the [documentation](#) to see how you can optimize CPU performance in your own project using the [C# Job System](#).

² These are part of the com.unity.collections package. Some of these structures are currently in Preview.

The Burst compiler

The [Burst compiler](#) complements the Job System. Burst translates IL/.NET bytecode into optimized native code using [LLVM](#). To access it, simply add the **com.unity.burst** package from the Package Manager.

Burst allows Unity developers to continue using a subset of C# for convenience while improving performance.

To enable the Burst compiler for your scripts:

- Remove static variables. If you need to write to a list, consider using a NativeArray decorated with the [NativeDisableContainerSafetyRestriction attribute](#). This allows parallel jobs to write to the NativeArray.
- Use [Unity.Mathematics](#) functions instead Mathf. functions.
- Decorate the job definition with the [BurstCompile attribute](#).

```
[BurstCompile]
public struct MyFirstJob : IJob
{
    public NativeArray<float3> ToNormalize;

    public void Execute()
    {
        for (int i = 0; i < ToNormalize.Length; i++)
        {
            ToNormalize[i] = math.normalize(ToNormalize[i]);
        }
    }
}
```

Here is an example Burst job that runs over an array of **float3**'s and normalizes the vectors. It uses the [Unity Mathematics](#) package, as mentioned above.

Both the C# Job System and the Burst compiler form part of Unity's [Data-Oriented Tech Stack \(DOTS\)](#). However, you can use them equally with 'classic' Unity GameObjects or the [Entity Component System](#). Refer to the [latest documentation](#) to see how Burst can accelerate your workflow when combined with the C# Job System.

Project configuration

There are a few project settings that can affect your performance.

Disable unnecessary Player or Quality settings

In the Player settings, disable Auto Graphics API and remove graphics APIs that you don't plan on supporting for each of your targeted platforms. This can prevent generating excessive shader variants. Disable Target Architectures for older CPUs if your application is not supporting them.

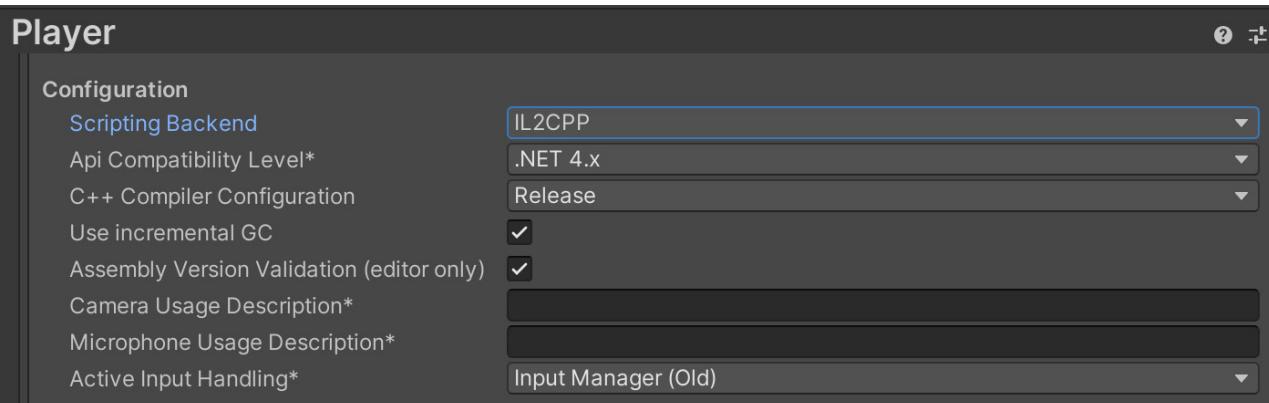
In the Quality settings, disable needless Quality levels.

Switch to IL2CPP

We recommend switching the Scripting Backend from Mono to IL2CPP (Intermediate Language to C++). Doing so will provide overall better runtime performance.

Be aware this does increase build times. Some developers prefer to use Mono locally for faster iteration, then switch to IL2CPP for build machines and/or release candidates. Refer to the [Optimizing IL2CPP build times](#) documentation to reduce your build times.

On PlayStation platforms where IL2CPP is the only option, locate the **Player Settings > Other Settings > IL2CPP optimization level** settings. Use the less optimal options during development to speed up build times. For profiling or final release, select **Optimized Compile, Remove Unused Code, Optimized link**.



Using this option, Unity converts IL code from scripts and assemblies to C++ before creating a native binary file (e.g., .exe, .apk, .xap) for your target platform.

Please refer to the [documentation](#), which provides information on how to optimize build times.

You can also read the [Introduction to IL2CPP Internals](#) blog post for additional detail or consult the [Compiler options manual page](#) to see how the various compiler options affect runtime performance.

Avoid large hierarchies

Split your hierarchies. If your GameObjects do not need to be nested in a hierarchy, simplify the parenting. Smaller hierarchies benefit from multithreading to refresh the Transforms in your scene. Complex hierarchies incur unnecessary Transform computations and more cost to garbage collection.

See [Optimizing the Hierarchy](#) and this [Unite talk](#) for best practices for Transforms.

Assets

The asset pipeline can dramatically impact your application's performance. An experienced technical artist can help your team define and enforce asset formats, specifications, and import settings for smooth processes.

Don't rely on default settings. Use the platform-specific override tab to optimize assets such as textures and mesh geometry. Incorrect settings might yield larger build sizes, longer build times, poor GPU performance, and poor memory usage. Consider using the [Presets](#) feature to help customize baseline settings that will enhance a specific project.

See [this guide to best practices for importing art assets](#). For a mobile-specific guide (with many general tips as well), check out the Unity Learn course on [3D art optimization for mobile applications](#).

Compress textures

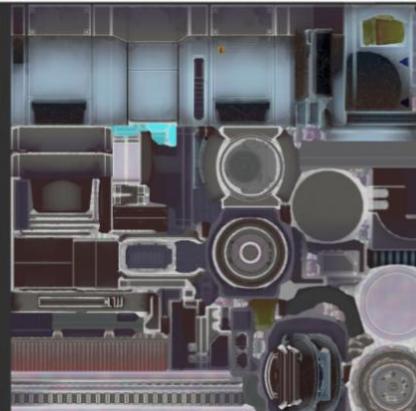
Consider these two examples using the same model and texture. The settings on the top consume more than five times the memory compared to those on the bottom, without much benefit in visual quality.

**uncompressed
texture (16MB)**



Uncompressed textures require more memory.

**DXT1
compressed (2.7MB)**



[Texture compression](#) offers significant performance benefits when you apply it correctly.

This can result in faster load times, a smaller memory footprint, and dramatically increased rendering performance. Compressed textures only use a fraction of the memory bandwidth needed for uncompressed 32-bit RGBA textures.

Refer to this [recommended list of texture compression formats](#) for your target platform:

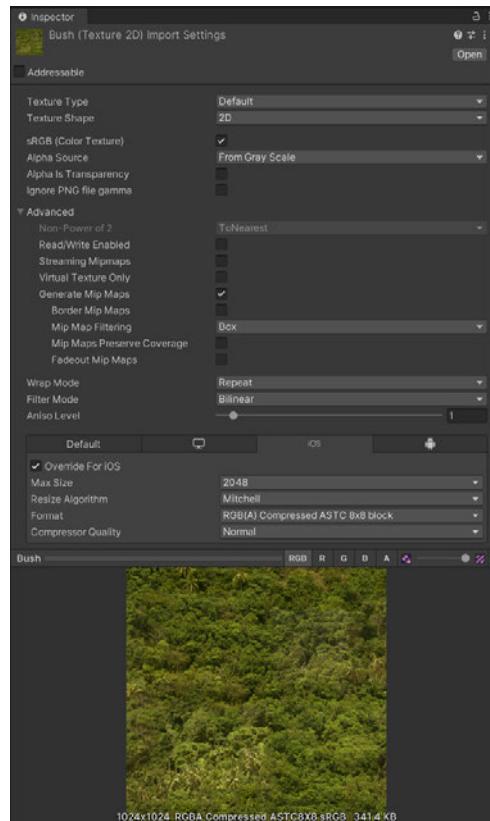
- **iOS / Android / Switch:** Use **ASTC**.
- **PC/XBox One/PS4: BC7** (high quality) or DXT1 (low/normal quality)

See the manual for more information on [recommended texture compression format by platform](#).

Texture import settings

Textures can potentially use a lot of resources. Import settings here are critical. In general, try to follow these guidelines:

- **Lower the Max Size:** Use the minimum settings that produce visually acceptable results. This is non-destructive and can quickly reduce your texture memory.
- **Use powers of two (POT):** Unity requires POT texture dimensions for texture compression formats.
- **Toggle off the Read/Write Enabled option:** When enabled, this option creates a copy in both CPU- and GPU-addressable memory, doubling the texture's memory footprint. In most cases, keep this disabled (only enable this if you generate a texture at runtime and need to overwrite it). You can also enforce this option via Texture2D.Apply, passing in makeNoLongerReadable set to true.
- **Disable unnecessary mipmaps:** Mipmaps are not needed for textures that remain at a consistent size on-screen, such as 2D sprites and UI graphics (leave mipmaps enabled for 3D models that vary their distance from the camera).



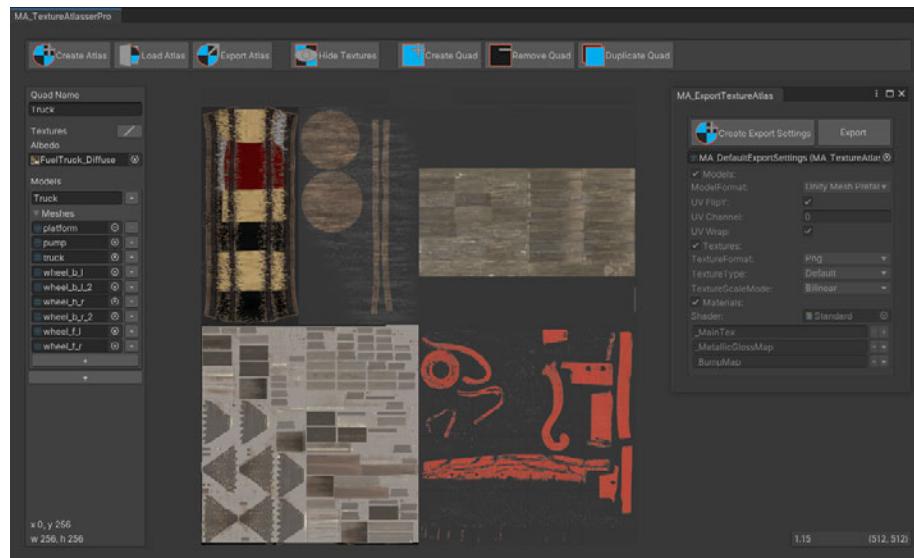
Proper texture import settings will help optimize your build size.

Atlas your textures

Atlasing is the process of grouping together several smaller textures into a single uniformly sized larger texture. This can reduce the GPU effort needed to draw the content (using fewer draw calls) and reduce memory usage.

For 2D projects, you can use a [Sprite Atlas](#) (**Asset > Create > 2D > Sprite Atlas**) rather than rendering individual Sprites and Textures.

For 3D projects, you can use your DCC package of choice. Several third-party tools like [MA_TextureAtlasser](#) or [TexturePacker](#) also can build texture atlases.



Use texture atlases to save draw calls.

Combine textures and remap UVs for any 3D geometry that doesn't require high-resolution maps. A visual editor gives you the ability to set and prioritize the sizes and positions in the texture atlas or sprite sheet.

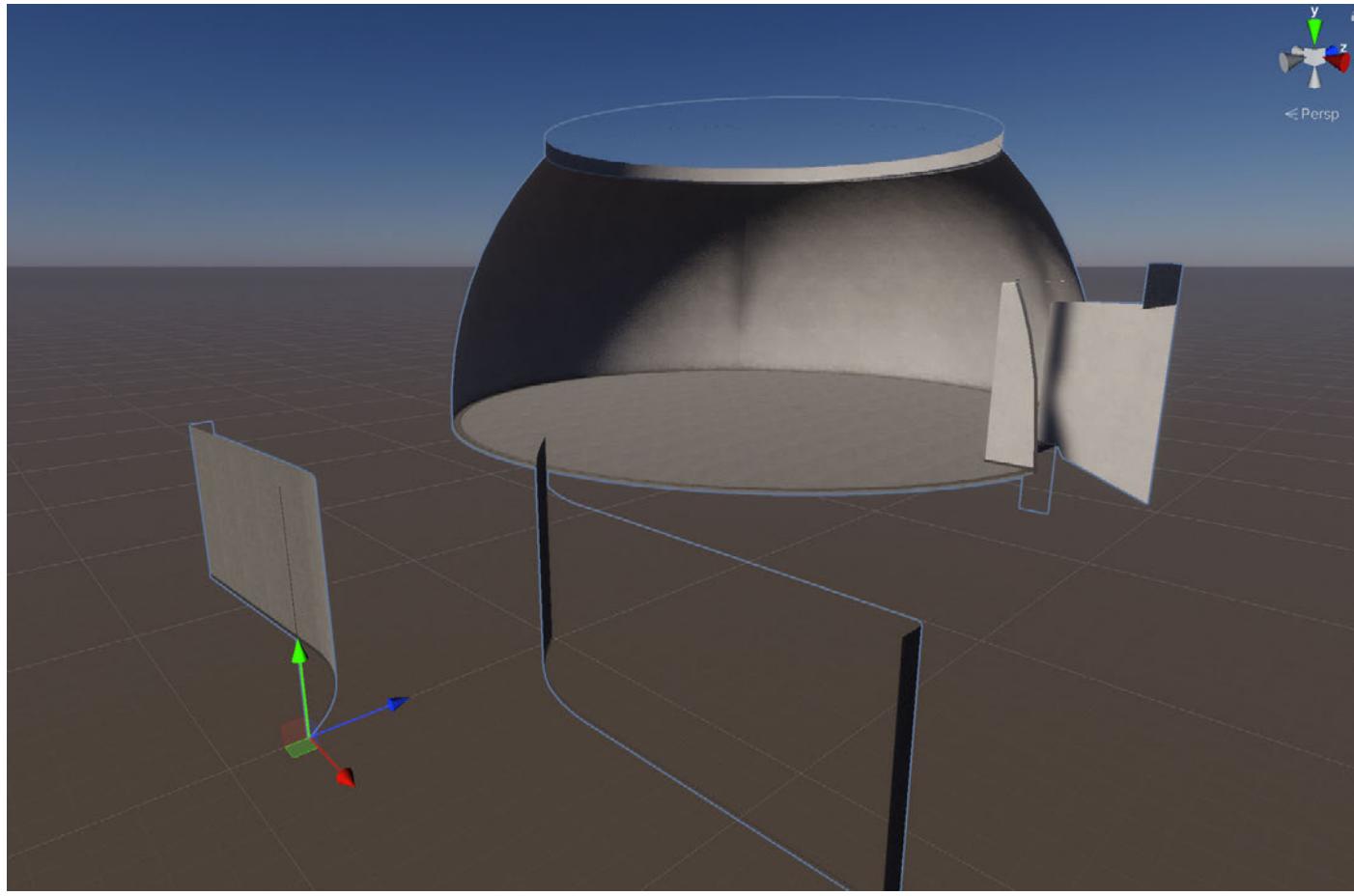
The texture packer consolidates the individual maps into one large texture. Unity can then issue a single draw call to access the packed Textures with a smaller performance overhead.

Check your polygon counts

Higher-resolution models mean more memory usage and potentially longer GPU times. Does your background geometry really need a million polygons?

Keep the geometric complexity of GameObjects in your Scenes to a minimum, otherwise Unity has to push a lot of vertex data to the graphics card.

Consider cutting down models in your DCC package of choice. Delete unseen polygons from the camera's point of view. For example, if you never see the back of a cupboard resting against a wall, the model should not have any faces there.



Remove any unseen faces to optimize your models.

Be aware that the bottleneck is not usually polygon *count* on modern GPUs, but rather polygon *density*. We recommend performing an art pass across all assets to reduce the polygon count of distant objects. [Microtriangles](#) can be a significant cause of poor GPU performance.

Depending on the target platform, investigate adding details via high-resolution Textures to compensate for low-poly geometry. Use textures and normal maps instead of increasing the density of the mesh.

Reduce pixel complexity by baking as much detail into the Textures as possible. For example, capture the specular highlights into the Texture to avoid having to compute the highlight in the fragment shader.

Be mindful and remember to profile regularly, as these techniques can impact performance, and may not be suitable for your target platform.

Mesh import settings

Much like textures, meshes can consume excess memory if not imported carefully. To minimize meshes' memory consumption:

- **Use mesh compression:** Aggressive mesh compression can reduce disk space (memory at runtime, however, is unaffected). Note that mesh quantization can result in inaccuracy, so experiment with compression levels to see what works for your models.
- **Disable Read/Write:** Enabling this option duplicates the mesh in memory, which keeps one copy of the mesh in system memory and another in GPU memory. In most cases, you should disable it (in Unity 2019.2 and earlier, this option is checked by default).
- **Disable rigs and BlendShapes:** If your mesh does not need skeletal or blendshape animation, disable these options wherever possible.
- **Disable normals and tangents:** If you are absolutely certain the mesh's material will not need normals or tangents, uncheck these options for extra savings.

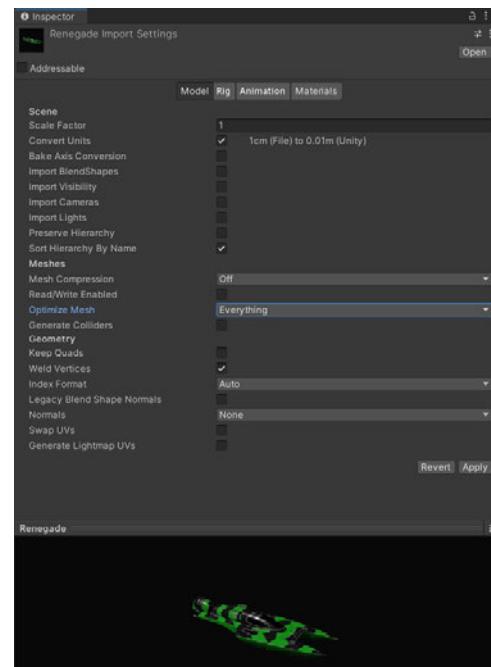
Other mesh optimizations

In the Player Settings, you can also apply a couple of other optimization to your meshes:

Vertex Compression sets vertex compression per channel. For example, you can enable compression for everything except positions and lightmap UVs. This can reduce runtime memory usage from your meshes.

Note that the Mesh Compression in each mesh's Import Settings *overrides* the vertex compression setting. In that event, the runtime copy of the mesh is uncompressed and may use more memory.

Optimize Mesh Data removes any data from meshes that is not required by the Material applied to them (such as tangents, normals, colors, and UVs).



Check your mesh import settings.

Audit your assets

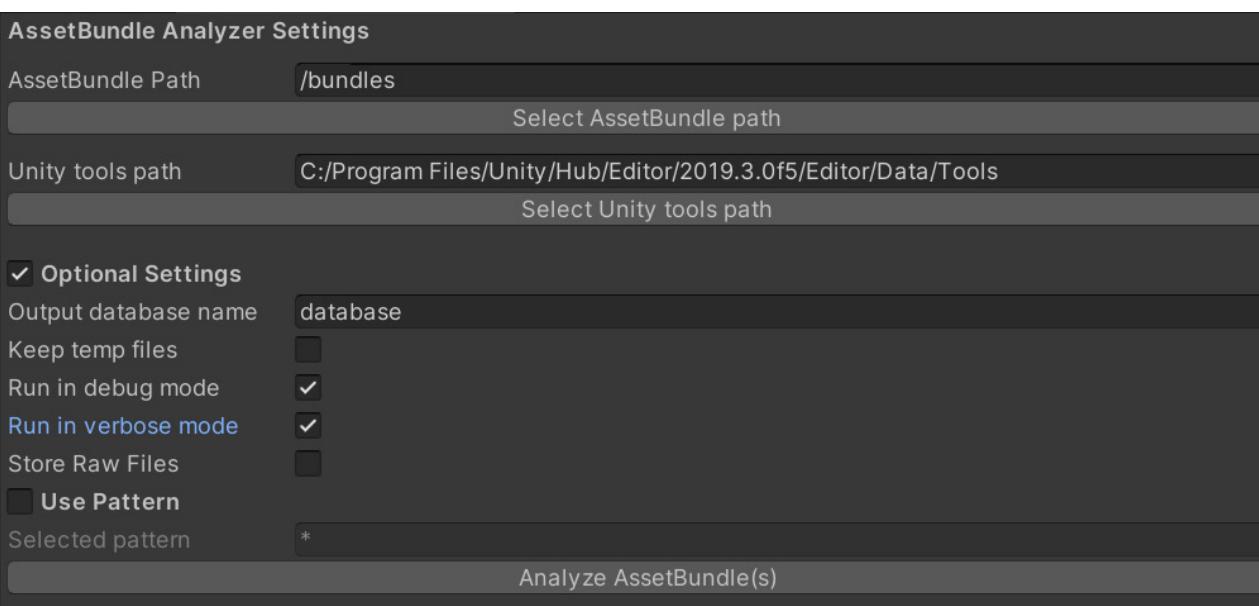
By automating the asset audit process, you can avoid accidentally changing asset settings. A couple tools can help both to standardize your import settings or analyze your existing assets.

The AssetPostprocessor

The [AssetPostprocessor](#) allows you to run scripts when importing assets. This prompts you to customize settings before and/or after importing models, textures, audio, etc.

Asset Bundle Analyzer

The [Asset Bundle Analyzer](#) (with a packaged version [here](#)) is a tool for reviewing your project. It runs a Python script that extracts information from Unity asset bundles, then stores that in SQLite database (.DB) in the project folder.



You can then use a tool like [DB Browser](#) to review the database for large assets, duplicates, shader variants, and read-write properties on textures and meshes.

Read more about [asset auditing](#) in the [Understanding Optimization in Unity](#) section of the best practice guide.

Async texture buffer

Unity uses a ring buffer to push textures to the GPU. You can manually adjust this async texture buffer via [QualitySettings.asyncUploadBufferSize](#).

If either the upload rate is too slow or the main thread stalls while loading several Textures at once, adjust the Texture [buffers](#). Usually you can set the value (in MB) to the size of the largest texture you need to load in the Scene.

Note: Be aware that changing the default values can lead to high memory pressure. Also, you cannot return ring buffer memory to the system after Unity allocates it. If GPU memory overloads, the GPU unloads the least-recently used Texture and forces the CPU to reupload it the next time it enters the camera frustum.

Read more about memory restrictions in Texture buffers when using time-slice awake in the [Memory Management in Unity](#) guide. Also, refer to the post [Optimizing loading performance](#) to investigate how you can improve your loading times with the Async Upload Pipeline.

Stream mipmaps and textures

The Mipmap Streaming system gives you control over which mipmap levels load into memory. To enable it, go to Unity's Quality Settings (**Edit > Project Settings > Quality**) and check **Texture Streaming**. Enable **Streaming Mipmaps** in the Texture's Import Settings under **Advanced**.

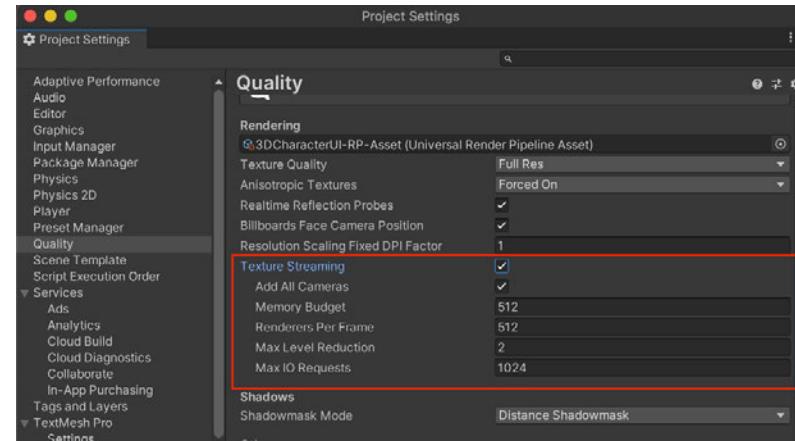
This system reduces the total amount of memory needed for Textures because it only loads the mipmaps necessary to render the current Camera position. Otherwise, Unity loads all of the textures by default. Texture Streaming trades a small amount of CPU resources to save a potentially large amount of GPU memory.

You can use the [Mipmap Streaming API](#) for additional control. Texture Streaming automatically reduces mipmap levels to stay within the user-defined Memory Budget.

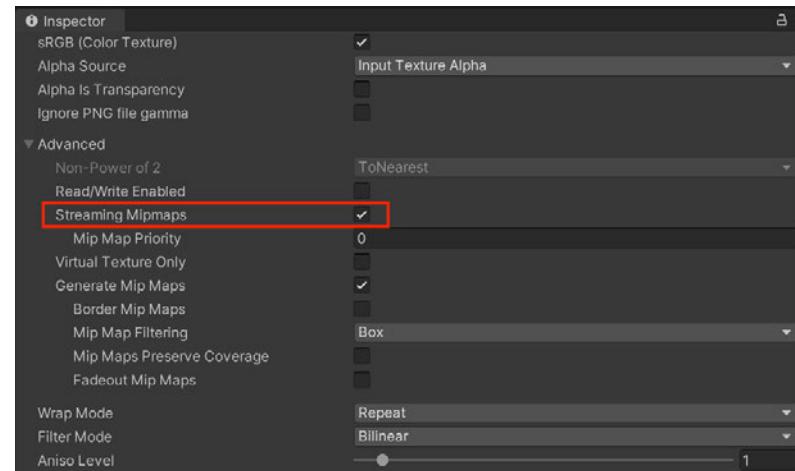
Use Addressables

The [Addressable Asset System](#) simplifies how you manage the assets that make up your game. Any asset, including Scenes, Prefabs, text assets, and so on, can be marked as "addressable" and given a unique name. You can then call this alias from anywhere.

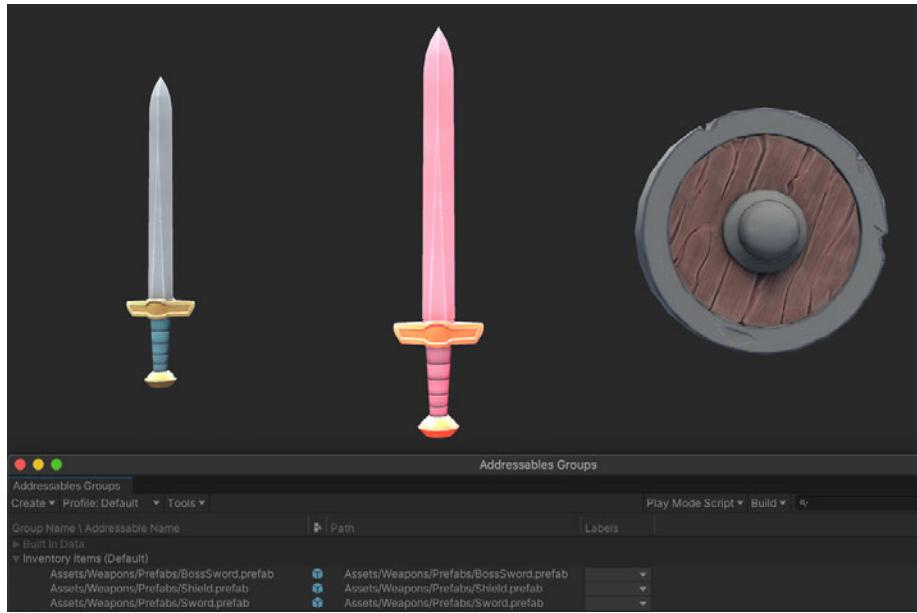
Adding this extra level of abstraction between the game and its assets can streamline certain tasks, such as creating a separate downloadable content pack. Addressables makes referencing those asset packs easier as well, whether they're local or remote.



Texture Streaming settings

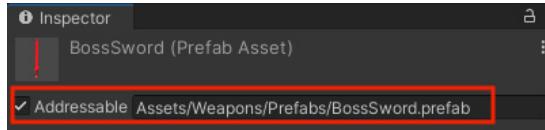


Streaming Mipmaps is enabled.



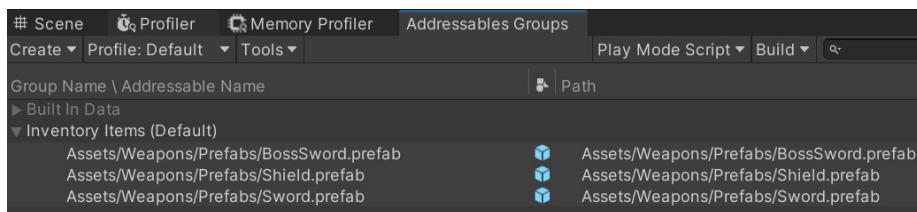
In this example, Addressables tracks the inventory of Prefabs.

Install the Addressables package from the Package Manager. Each asset or Prefab in the project has the ability to become “addressable” as a result. Checking the option under an asset’s name in the Inspector assigns it a default unique address.



Addressable option enabled with default Addressable Name

Once marked, the corresponding assets appear in the
Window > Asset Management > Addressables > Groups window.



In the Addressables Groups, you can see each asset’s custom address paired with its location.

Whether the asset is hosted elsewhere or stored locally, the system will locate it using the Addressable Name string. An addressable Prefab does not load into memory until needed and automatically unloads its associated assets when no longer in use.

The [Tales from the Optimization Trenches: Saving Memory with Addressables](#) blog post demonstrates an example of how to organize your Addressable Groups in order to be more efficient with memory. See also [Addressables: Introduction to Concepts](#) for a quick overview of how the Addressable Asset system can work in your project.

Graphics

Unity's graphics tools let you create beautiful, optimized graphics across a range of platforms, from mobile to high-end consoles and desktop. Because lighting and effects are quite complex, we recommend that you thoroughly review the [render pipeline documentation](#) before attempting to optimize.

Commit to a render pipeline

Optimizing scene lighting is not an exact science. Your process usually depends on your artistic direction and render pipeline.

Before you begin lighting your scenes, you will need to choose one of the available render pipelines. A render pipeline performs a series of operations that take the contents of a Scene to display them on-screen.

Unity provides three prebuilt render pipelines with different capabilities and performance characteristics, or you can create your own.

- The [Built-in Render Pipeline](#) is a general-purpose render pipeline with limited customization. This is the default.
- The [Universal Render Pipeline \(URP\)](#) is a prebuilt [Scriptable Render Pipeline](#). URP provides artist-friendly workflows to create optimized graphics across a range of platforms, from mobile to high-end consoles and PCs.

URP adds graphics and rendering features unavailable to the Built-in Render Pipeline. In order to maintain performance, it makes tradeoffs to reduce the computational cost of lighting and shading. Choose URP if you want to reach the most target platforms, including mobile and VR.

- The [High Definition Render Pipeline \(HDRP\)](#) is another prebuilt [Scriptable Render Pipeline](#), designed for cutting-edge, high-fidelity graphics.

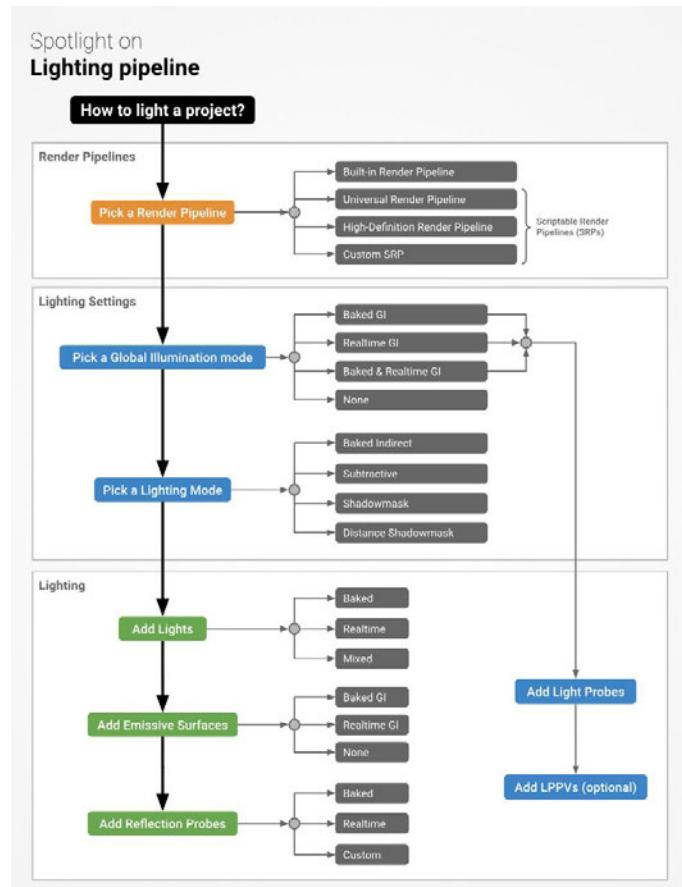
HDRP targets high-end hardware such as PC, Xbox, and PlayStation.³ Use it to create realistic games, automotive demos, or architectural applications. HDRP uses physically based Lighting and Materials and supports improved debugging tools.

³HDRP is not currently supported on mobile platforms or Nintendo Switch. See the [Requirements and compatibility page](#) for more details.

Though the Built-in Render Pipeline is not as customizable as URP or HDRP, it does support a wide number of platforms. To use it, you'll need to configure the different [rendering paths](#), and extend its functionality with [command buffers](#) and [callbacks](#).

URP and HDRP work on top of the Scriptable Render Pipeline (SRP). This is a thin API layer that lets you schedule and configure rendering commands using C# scripts. This flexibility allows you to customize virtually every part of the pipeline. You can also create your own [custom render pipeline](#) based on SRP.

See [Render Pipelines in Unity](#) for a more detailed comparison of the available pipelines.



Choose a render pipeline early when planning your project.



[The Heretic](#) short film showcases HDRP's high-end graphical capabilities.

Render pipeline packages for consoles

To build a Project for the **PlayStation 4**, **PlayStation 5**, **Xbox Series**, **Game Core Xbox One** or **Xbox One**, you need to install an additional package for each platform you want to support. The packages for each platform are:

- **PlayStation 4:** com.unity.render-pipelines.ps4
- **PlayStation 5:** com.unity.render-pipelines.ps5
- **Xbox One:** com.unity.render-pipelines.xboxone
- **Game Core Xbox Series:** com.unity.render-pipelines.gamecore
- **Game Core Xbox One:** com.unity.render-pipelines.gamecore

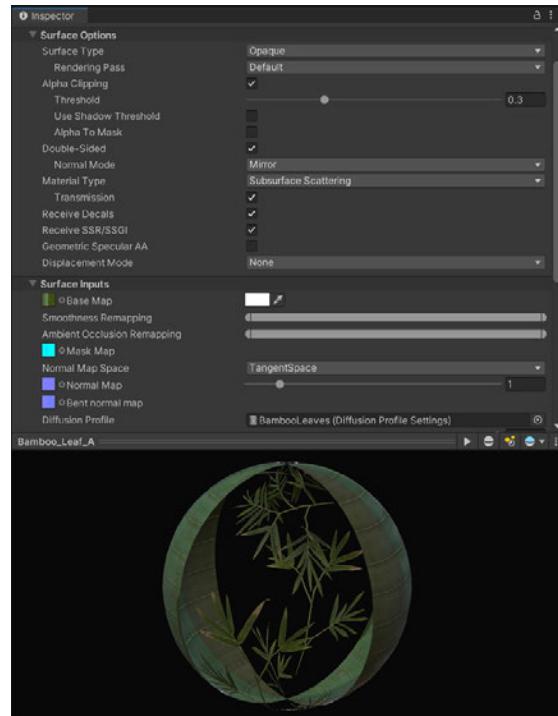
Select a rendering path

While selecting a render pipeline, you should also consider a *rendering path*. The rendering path represents a specific series of operations related to lighting and shading. Deciding on a rendering path depends on your application needs and target hardware.

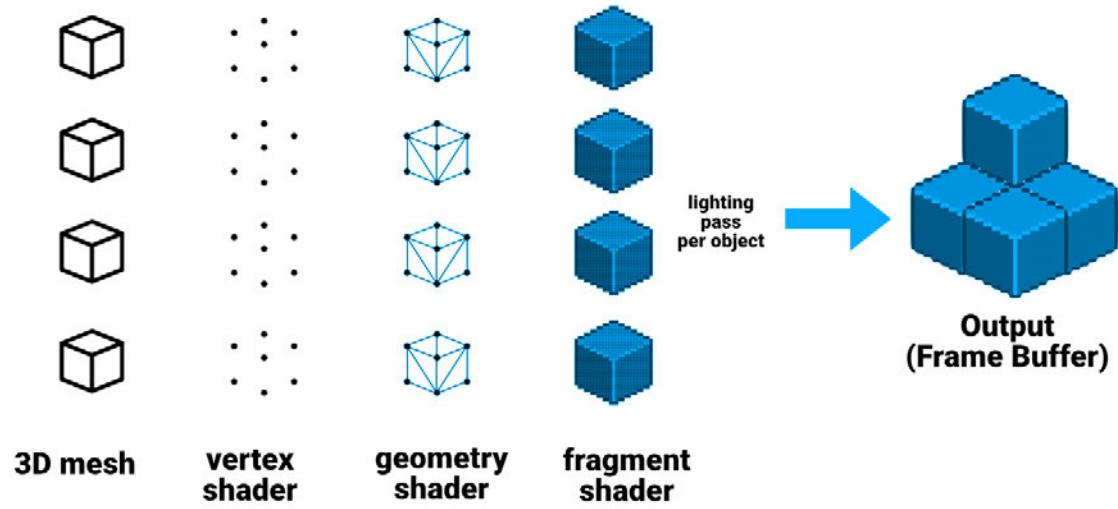
Forward rendering path

URP and the Built-in Render Pipeline both use forward renderers. In forward rendering, the graphics card projects the geometry and splits it into vertices. Those vertices are further broken down into fragments, or pixels, which render to screen to create the final image.

The pipeline passes each object, one at a time, to the graphics API. Forward rendering comes with a cost for each light. The more lights in your Scene, the longer rendering will take.



Materials, such as skin or foliage, can benefit from the advanced lighting and shading features preconfigured with the HDRP.

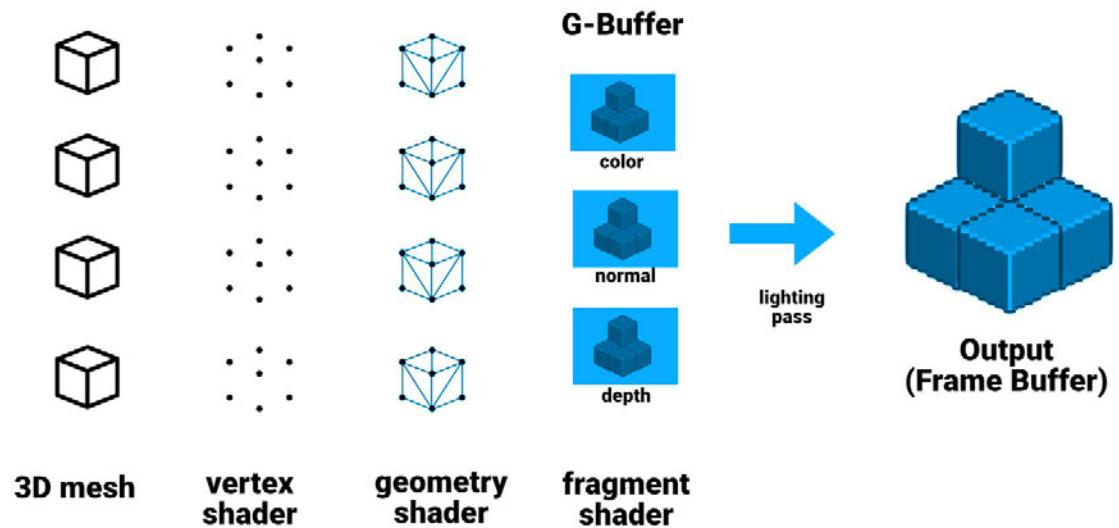


The Built-in Render Pipeline's forward renderer draws each light in a separate pass per object. If you have multiple lights hitting the same GameObject, this can create significant overdraw, where overlapping areas need to draw the same pixel more than once. Minimize the number of real-time lights to reduce overdraw.

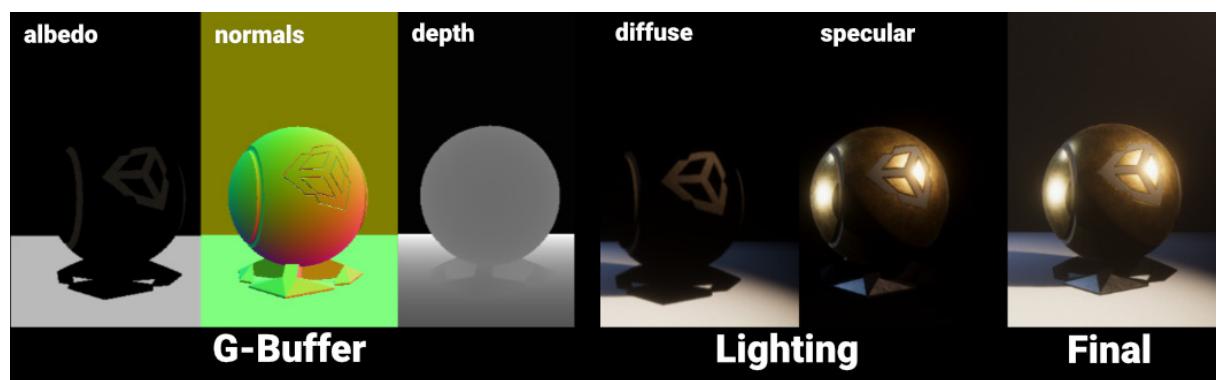
Rather than rendering one pass per light, the URP culls the lights per-object. This allows for the lighting to be computed in one single pass, resulting in fewer draw calls compared to the Built-In Render Pipeline's forward renderer.

Deferred shading path

The Built-in Render Pipeline and the HDRP can also use deferred shading.
In deferred shading, lighting is not calculated per object.



Deferred shading path



Deferred shading applies lighting to a buffer instead of each object.

Deferred shading instead postpones heavy rendering – like lighting – to a later stage. Deferred shading uses two passes.

In the first pass, or the [G-Buffer](#) geometry pass, Unity renders the GameObjects. This pass retrieves several types of geometric properties and stores them in a set of textures. G-buffer textures can include:

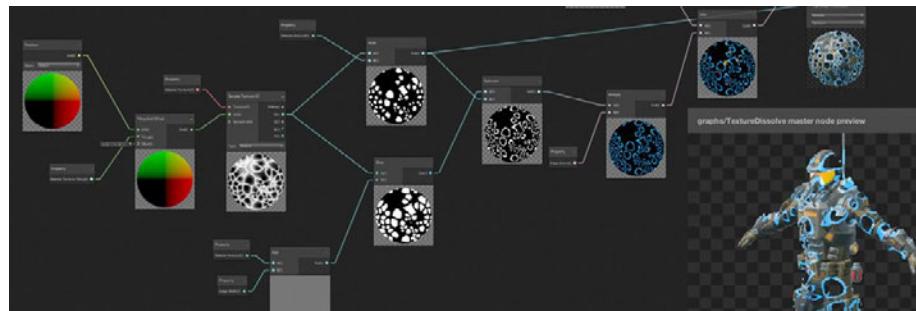
- diffuse and specular colors
- surface smoothness
- occlusion
- world space normals
- emission + ambient + reflections + lightmaps

In the second pass, or [lighting pass](#), Unity renders the Scene's lighting based on the G-buffer. Imagine iterating over each pixel and calculating the lighting information based on the buffer instead of the individual objects. Thus, adding more non-shadow casting lights in deferred shading does not incur the same performance hit as with forward rendering.

Though choosing a rendering path is not an optimization per se, it can affect *how* you optimize your project. The other techniques and workflows in this section may vary depending on what render pipeline and which rendering path you've chosen.

Optimize Shader Graphs

Both HDRP and URP support [Shader Graph](#), a visual interface for shader creation. This allows some users to create complex shading effects that may have been previously out of reach. Use the 150+ nodes in the visual graph system to create more shaders. You can also make your own custom nodes with the API.



Build shaders using a visual interface.

Begin each Shader Graph with a compatible master node, which determines the graph's output. Add nodes and operators with the visual interface, and construct the shader logic.

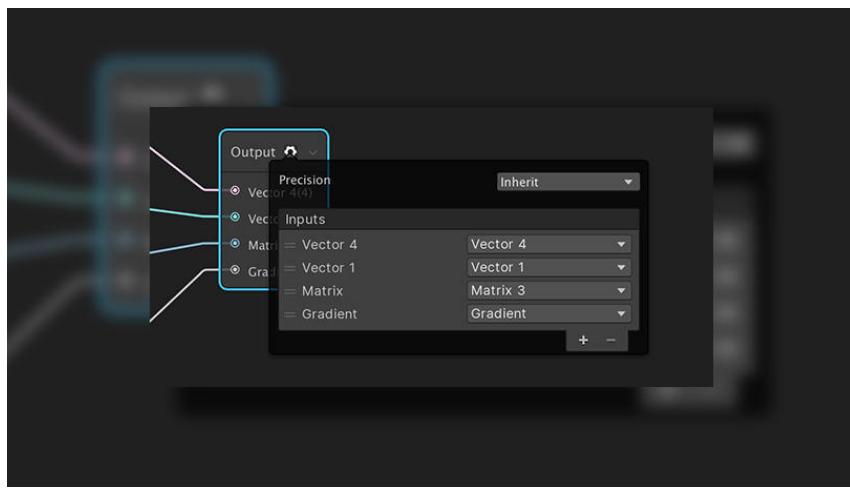
This Shader Graph then passes into the render pipeline's backend. The final result is a ShaderLab shader, functionally similar to one written in HLSL or Cg.

Optimizing a Shader Graph follows many of the same rules that apply to traditional HLSL/Cg Shaders. The more processing your Shader Graph does, the more it will impact the performance of your application.

If you are CPU-bound, optimizing your shaders won't improve frame rate, but may improve your battery life for mobile platforms.

If you are GPU-bound, follow these guidelines for improving performance with Shader Graphs:

- **Decimate your nodes:** Remove unused nodes. Don't change any defaults or connect nodes unless those changes are necessary. Shader Graph compiles out any unused features automatically.
When possible, bake values into textures. For example, instead of using a node to brighten a texture, apply the extra brightness into the texture asset itself.
- **Use a smaller data format:** Switch to a smaller data structure when possible. Consider using Vector2 instead of Vector3 if it does not impact your project. You can also reduce precision if the situation allows (e.g., **half** instead of **float**).



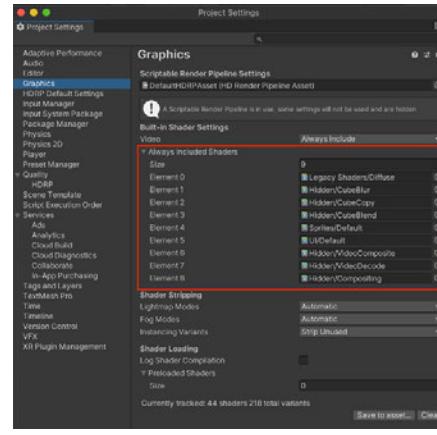
Reduce precision in Shader Graph in the Output node when possible.

- **Reduce math operations:** Shader operations run many times per second, so optimize any math operators when possible. Try to blend results instead of creating a logical branch. Use constants, and combine scalar values before applying vectors. Finally, convert any properties that do not need to appear in the Inspector as in-line Nodes. All of these incremental speedups can help your frame budget.
- **Branch a preview:** As your graph gets larger, it may become slower to compile. Simplify your workflow with a separate, smaller branch just containing the operations you want to preview at the moment, then iterate more quickly on this smaller branch until you achieve the desired results.
If the branch is not connected to the master node, you can safely leave the preview branch in your graph. Unity removes nodes that do not affect the final output during compilation.
- **Manually optimize:** Even if you're an experienced graphics programmer, you can still use a Shader Graph to lay down some boilerplate code for a script-based shader. Select the Shader Graph asset, then select Copy Shader from the context menu.
Create a new HLSL/Cg Shader and then paste in the copied Shader Graph. This is a one-way operation, but it lets you squeeze additional performance with manual optimizations.

Remove built-in shader settings

Remove every shader that you don't use from the Always Included list of shaders in the Graphics Settings ([Edit > ProjectSettings > Graphics](#)). Add shaders here needed for the lifetime of the application.

Create a new HLSL/Cg Shader and then paste in the copied Shader Graph. This is a one-way operation, but it lets you squeeze additional performance with manual optimizations.



Always Included Shaders

Strip shader variants

You can use the [Shader compilation pragma directives](#) to compile the shader differently for target platforms. Then, use a shader keyword (or [Shader Graph Keyword](#) node) to create [shader variants](#) with certain features enabled or disabled.

Shader variants can be useful for platform-specific features but increase build times and file size. You can prevent shader variants from being included in your build, if you know that they are not required.

Parse the [Editor.log](#) for shader timing and size. Locate the lines that begin with "Compiled shader" and "Compressed shader."

In an example log, your TEST shader may show you:

```
Compiled shader 'TEST Standard (Specular setup)' in 31.23s
    d3d9 (total internal programs: 482, unique: 474)
    d3d11 (total internal programs: 482, unique: 466)
    metal (total internal programs: 482, unique: 480)
    glcore (total internal programs: 482, unique: 454)
Compressed shader 'TEST Standard (Specular setup)' on d3d9 from 1.04MB to 0.14MB
Compressed shader 'TEST Standard (Specular setup)' on d3d11 from 1.39MB to 0.12MB
Compressed shader 'TEST Standard (Specular setup)' on metal from 2.56MB to 0.20MB
Compressed shader 'TEST Standard (Specular setup)' on glcore from 2.04MB to 0.15MB
```

This tells you a few things about this shader:

- The shader expands into 482 variants due to #pragma multi_compile and shader_feature.
- Unity compresses the shader included in the game data to roughly the sum of the compressed sizes: $0.14+0.12+0.20+0.15 = 0.61\text{MB}$.
- At runtime, Unity keeps the compressed data in memory (0.61MB), while the data for your currently used graphics API is uncompressed. For example, if your current API was Metal, that would account for 2.56MB.

After a build, [Project Auditor](#) can parse the Editor.log to display a list of all shaders, shader keywords, and shader variants compiled into a project. It can also analyze the [Player.log](#) after the game is run. This shows you what variants the application actually compiled and used at runtime.

Employ this information to build a scriptable shader stripping system and reduce the number of variants. This can improve build times, build sizes, and runtime memory usage.

Read the [Stripping scriptable shader variants](#) blog post to see this process in detail.

Particle simulations: Particle System or Visual Effect Graph

Unity includes two particle simulation solutions for smoke, liquids, flames, or other effects:

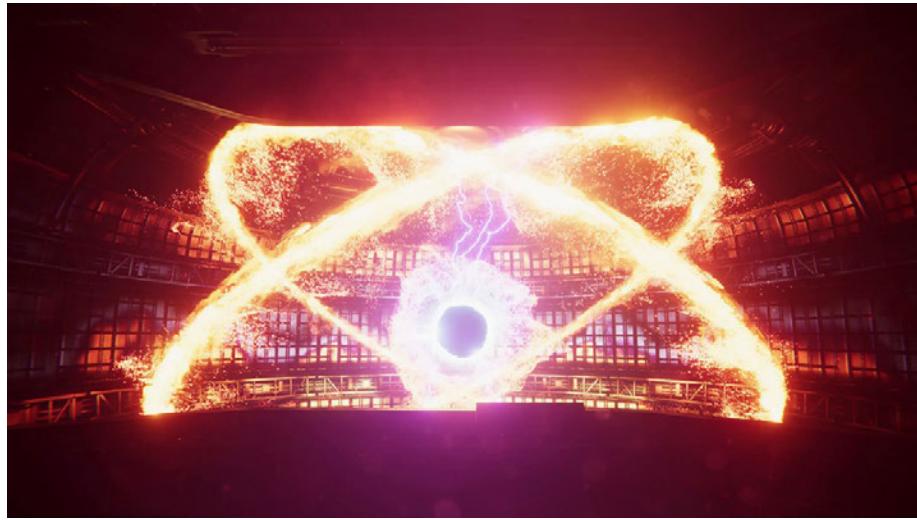
- The [Particle System](#) can simulate thousands of particles on the CPU. You can use C# scripts to define a system and its individual particles. Particle Systems can interact with Unity's underlying physics system and any Colliders in your Scene. Particle Systems offer maximum compatibility and work with any of Unity's supported build platforms.



A simple effects simulation using the Particle System

- The [Visual Effect Graph](#) moves calculations on the GPU using compute shaders. This can simulate millions of particles in large-scale visual effects. The workflow includes a highly customizable graph view. Particles can also interact with the color and depth buffer.

Though it does not have access to the underlying physics system, a Visual Effect Graph can interact with complex assets, such as Point Caches, Vector Fields, and Signed Distance Fields. Visual Effect Graph only works on [platforms that support compute shaders](#) and HDRP (support for URP is currently in Preview).



Millions of particles on-screen created with the Visual Effect Graph

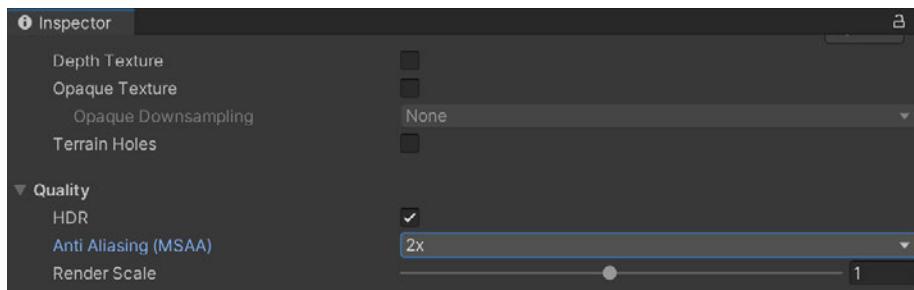
When selecting one of the two systems, keep device compatibility in mind. Most PCs and consoles support compute shaders, but many mobile devices do not. If your target platform does support [compute shaders](#), Unity allows you to use both types of particle simulation in your project.

Smooth with anti-aliasing

Anti-aliasing is highly desirable as it helps to smooth the image, reduce jagged edges, and minimize specular aliasing.

If you are using Forward Rendering with the Built-in Render Pipeline, [Multisample Anti-aliasing \(MSAA\)](#) is available in the [Quality Settings](#). MSAA produces high-quality anti-aliasing, but it can be expensive. The **MSAA Sample Count** from the drop-down menu (None, 2X, 4X, 8X) defines how many samples the renderer uses to evaluate the effect.

If you are using Forward Rendering with the URP or HDRP, you can enable MSAA on the Render Pipeline Asset.



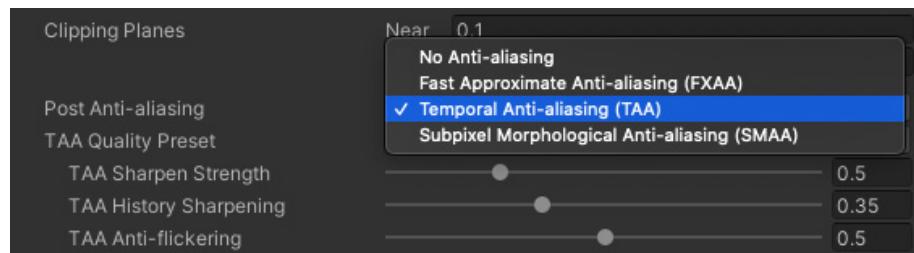
In URP, locate the MSAA settings on the Render Pipeline Asset.

Alternatively, you have the option to add anti-aliasing as a post-processing effect. This appears on the Camera component under **Anti-aliasing**:

- **Fast approximate anti-aliasing (FXAA)** smooths edges on a per-pixel level. This is the least resource intensive anti-aliasing and slightly blurs the final image.
- **Subpixel morphological anti-aliasing (SMAA)** blends pixels based on the borders of an image. This has much sharper results than FXAA and is suited for flat, cartoon-like, or clean art styles.

In HDRP, you can also use FXAA and SMAA in the **Post Anti-aliasing** on the Camera. HDRP also offers an additional option:

- **Temporal anti-aliasing (TAA)** smooths edges using frames from the history buffer. This works more effectively than FXAA but requires [motion vectors](#) in order to work. TAA can also improve Ambient Occlusion and Volumetrics. It is generally higher quality than FXAA, but it costs more resources and can produce occasional ghosting artifacts.



Anti-aliasing as a post effect on an HDRP camera.

Common lighting optimizations

While lighting is a vast subject, these general tips can help you to optimize your resources.

Bake lightmaps

The fastest option to create lighting is one that doesn't need to be computed per-frame. To do this, use [Lightmapping](#) to "bake" static lighting just once, instead of calculating it in real-time.

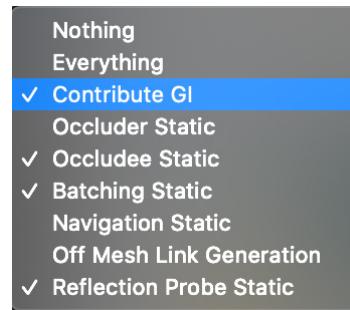
Add dramatic lighting to your static geometry using **Global Illumination (GI)**. Mark objects with **Contribute GI** so you can store high-quality lighting in the form of Lightmaps.

The process of generating a lightmapped environment takes longer than just placing a light in the scene in Unity, but this:

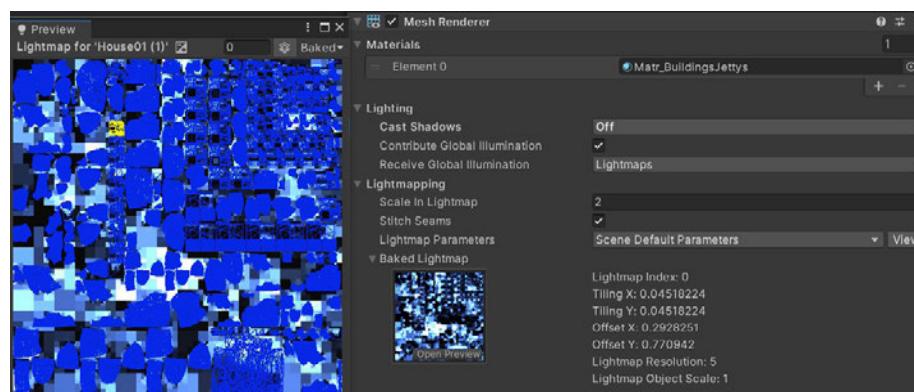
- runs faster, 2–3 times faster for two-per-pixel lights
- looks better – Global Illumination can calculate realistic-looking direct and indirect lighting. The lightmapper smooths and denoises the resulting map.

Baked shadows and lighting can then render without the same performance hit of real-time lighting and shadows.

Complex Scenes may require long bake times. If your hardware supports the **Progressive GPU Lightmapper**, this option can dramatically speed up your lightmap generation, up to tenfold in some cases.



Enable Contribute GI



Adjust the Lightmapping Settings ([Windows > Rendering > Lighting Settings](#)) and Lightmap size to limit memory usage.

Follow the [manual guide](#) and [this article on optimizing lighting](#) to get started with lightmapping in Unity.

Minimize Reflection Probes

A [Reflection Probe](#) can create realistic reflections, but this can be very costly in terms of batches. Use low-resolution cubemaps, culling masks, and texture compression to improve runtime performance. Use **Type: Baked** to avoid per-frame updates.

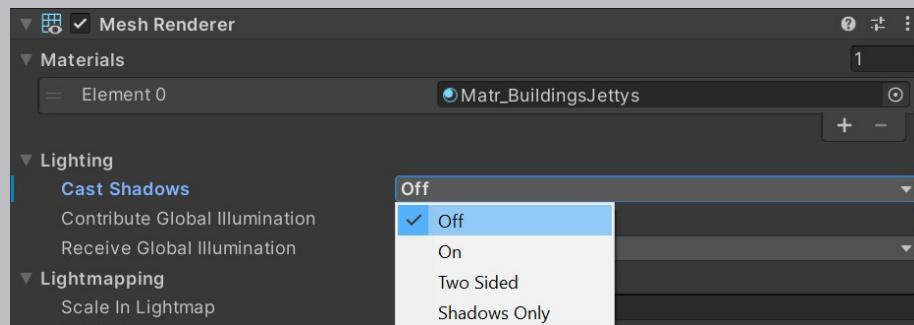
If using **Type: Realtime** is necessary in URP, avoid **Every Frame** if possible. Adjust the [Refresh Mode](#) and [Time Slicing](#) settings to reduce the update rate. You can also control the refresh with the Via Scripting option and [render the probe](#) from a custom script.

If using **Type: Realtime** is necessary in HDRP, use **On Demand** mode. You can also modify the Frame Settings in **Project Settings > HDRP Default Settings**. Reduce the quality and features under Realtime Reflection for improved performance.

Disable shadows

Shadow casting can be disabled per MeshRenderer and light. Disable shadows whenever possible to reduce draw calls.

You can also create fake shadows using a blurred texture applied to a simple mesh or quad underneath your characters. Otherwise, you can create blob shadows with custom shaders.



Disable shadow casting to reduce draw calls.

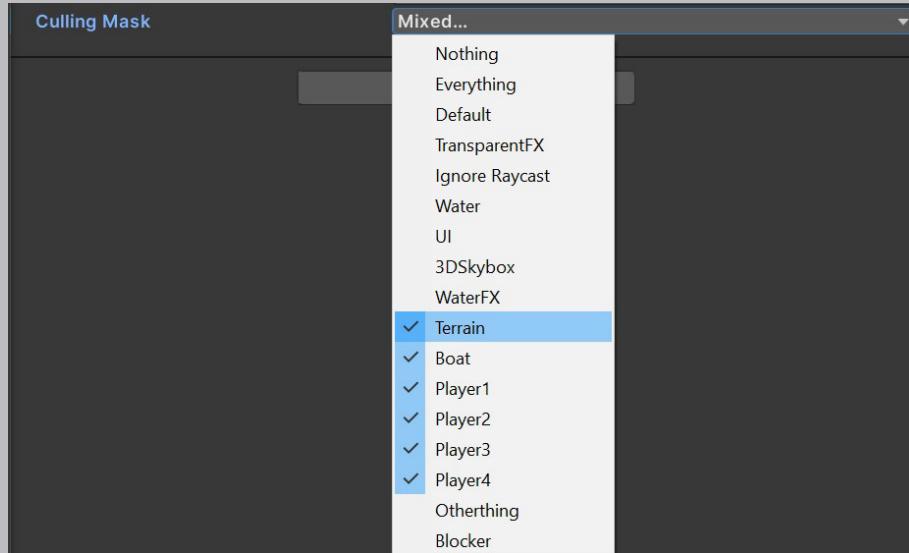
In particular, avoid enabling shadows for point lights. Each point light with shadows requires six shadow map passes per light – compare that with a single shadow map pass for a spotlight. Consider replacing point lights with spotlights where dynamic shadows are absolutely necessary. If you can avoid dynamic shadows, use a cubemap as a [Light.cookie](#) with your point lights instead.

Substitute a shader effect

In some cases, you can apply simple tricks rather than adding multiple extra lights. For example, instead of creating a light that shines straight into the camera to give a rim lighting effect, use a Shader which simulates rim lighting (see [Surface Shader examples](#) for an implementation of this in HLSL).

Use Light Layers

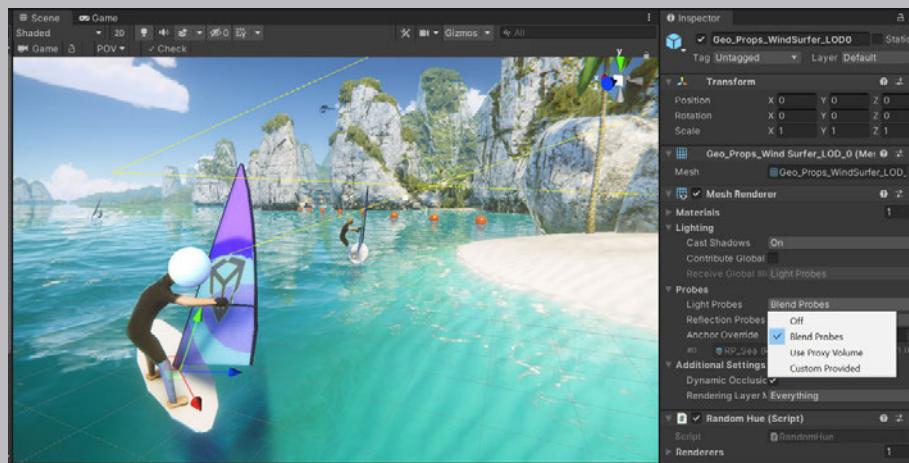
For complex scenes with multiple lights, separate your objects using layers, then confine each light's influence to a specific culling mask.



Layers can limit your light's influence to a specific culling mask.

Use Light Probes for moving or background objects

Light Probes store baked lighting information about the empty space in your Scene, while providing high-quality lighting (both direct and indirect). They use [Spherical Harmonics](#), which calculate very quickly compared to dynamic lights. This is especially useful for moving objects, which normally cannot receive baked lightmapping.



Light Probes illuminate dynamic objects in the background.

Light Probes can apply to static meshes as well. In the MeshRenderer component, locate the **Receive Global Illumination** dropdown and toggle it from **Lightmaps** to **Light Probes**.

Continue using lightmapping for your prominent level geometry, but switch smaller details to probe lighting. Light Probe illumination does not require proper UVs, saving you the extra step of unwrapping your meshes. Probes also reduce disk space since they don't generate lightmap textures.



You can also use Light Probes for smaller details where lightmapping is less noticeable.



A Light Probe Group with Light Probes spread across the level.

See the [Static Lighting with Light Probes](#) blog post for information about selectively lighting scene objects with [Light Probes](#).

For more about lighting workflows in Unity, read [Making believable visuals in Unity](#).

GPU optimization

To effectively optimize your graphics rendering, you'll need to understand the limitations of your target hardware and how to profile the GPU. Profiling helps you check and verify that the optimizations you're making are effective.

Use these best practices for reducing the rendering workload on the GPU.

Benchmark the GPU

When profiling, it's useful to start with a benchmark. A benchmark tells you what profiling results you should expect from specific GPUs.

See [GFXBench](#) for a great list of different industry-standard benchmarks for GPUs and graphics cards. The website provides a good overview of the current GPUs available and how they stack up against each other.

Watch the rendering statistics

Click the **Stats** button in the top right of the Game view. This window shows you real-time rendering information about your application during Play mode. Use this data to help optimize performance:

- **FPS:** Frames per second
- **CPU Main:** Total time to process one frame
(and update the Editor for all windows)
- **CPU Render:** Total time to render one frame of the Game view
- **Batches:** Groups of draw calls to be drawn together
- **Tris (triangles) and Verts (vertices):** Mesh geometry
- **SetPass calls:** The number of times Unity must switch shader passes to render the GameObjects on-screen; each pass can introduce extra CPU overhead.

Note: In-Editor fps does not necessarily translate to build performance. We recommend that you profile your build for the most accurate results. Frame time in milliseconds is a [more accurate metric than frames per second](#) when benchmarking, as outlined in the [“Frames per second: A deceptive metric”](#) section above.

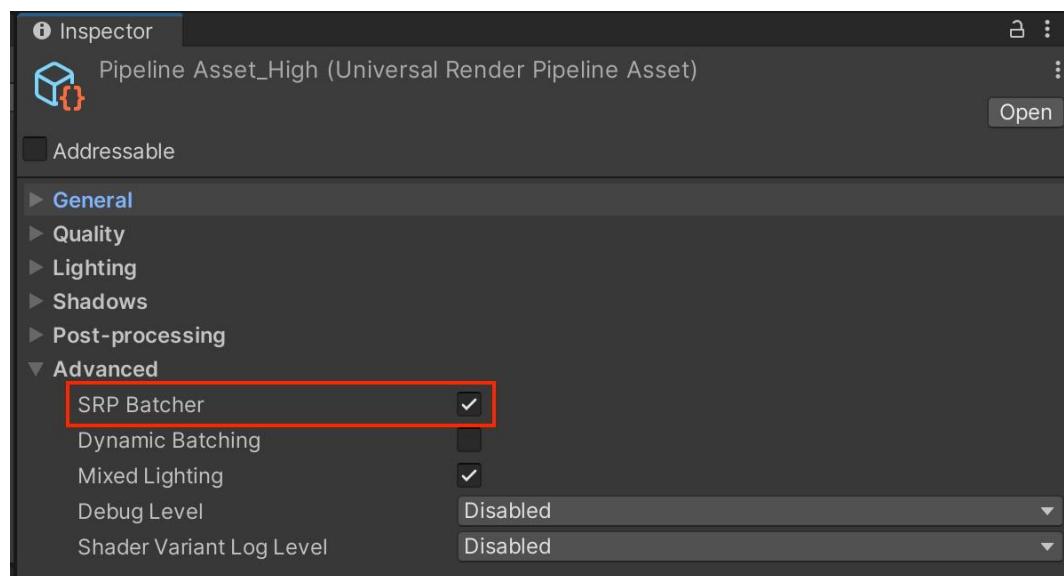
Use draw call batching

To draw a GameObject onscreen, Unity issues a draw call to the graphics API (e.g., OpenGL, Vulkan, or Direct3D). Each draw call is resource intensive. State changes between draw calls, such as switching Materials, can cause performance overhead on the CPU side.

PC and 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](#).

Draw call batching minimizes these state changes and reduces the CPU cost of rendering objects. Unity can combine multiple objects into fewer batches using several techniques:

- **SRP Batching:** If you are using HDRP or URP, enable the [SRP Batcher](#) in your Pipeline Asset under **Advanced**. When using compatible shaders, the SRP Batcher reduces the GPU setup between DrawCalls and makes material data persistent in GPU Memory. This can speed up your CPU rendering times significantly. Use fewer [Shader Variants](#) with a minimal amount of Keywords to improve SRP batching. Consult [this SRP documentation](#) to see how your project can take advantage of this rendering workflow.



SRP Batcher helps you [batch draw calls](#).

- **GPU instancing:** If you have a large number of identical objects (e.g., buildings, trees, grass, etc. with the same mesh and Material), use [GPU instancing](#). This technique batches them using graphics hardware. To enable GPU Instancing, select your Material in the Project window and in the Inspector, check **Enable Instancing**.
- **Static batching:** For non-moving geometry, Unity can reduce draw calls for any meshes sharing the same material. It is more efficient than dynamic batching, but it uses more memory.

Mark all meshes that never move as **Batching Static** in the Inspector. Unity combines all static meshes into one large mesh at build time. The [StaticBatchingUtility](#) also allows you to create these static batches yourself at runtime (for example, after generating a procedural level of non-moving parts).

- **Dynamic Batching:** For small meshes, Unity can group and transform vertices on the CPU, then draw them all in one go. Note: Do *not* use this unless you have enough low-poly meshes (no more than 300 vertices each and 900 total vertex attributes). Otherwise, enabling it will waste CPU time looking for small meshes to batch.

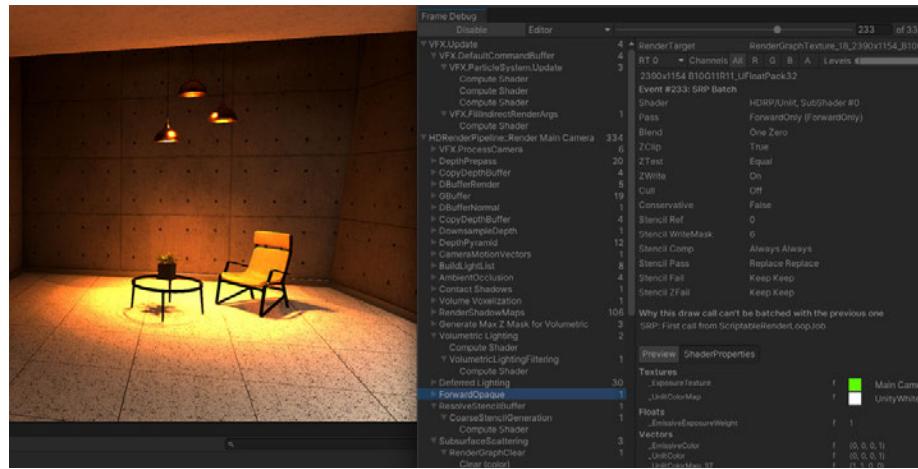
You can maximize batching with a few simple rules:

- Use as few Textures in a Scene as possible. Fewer Textures require fewer unique Materials, making them easier to batch. Additionally, use Texture atlases wherever possible.
- Always bake lightmaps at the largest atlas size possible. Fewer lightmaps require fewer Material state changes, but keep an eye on the memory footprint.
- Be careful not to instance Materials unintentionally. Accessing [Renderer.material](#) in scripts duplicates the material and returns a reference to the new copy. This breaks any existing batch that already includes the material. If you wish to access the batched object's material, use [Renderer.sharedMaterial](#) instead.
- Keep an eye on the number of static and dynamic batch counts versus the total draw call count by using the Profiler or the rendering stats during optimizations.

Please refer to the [Draw Call Batching](#) documentation for more information.

Check the Frame Debugger

The **Frame Debugger** allows you to freeze playback on a single frame and step through how Unity constructs a Scene to identify optimization opportunities. Look for GameObjects that render unnecessarily, and disable those to reduce draw calls per frame.



The Frame Debugger breaks down each rendered frame.

Note: The Frame Debugger does not show individual draw calls or state changes. Only native GPU profilers give you detailed draw call and timing information. However, the Frame Debugger can still be very helpful in debugging pipeline problems or batching issues.

One advantage of the Unity Frame Debugger is that you can relate a draw call to a specific GameObject in the Scene. This makes it easier to investigate certain issues that may not be possible in external frame debuggers.

For more information, read the [Frame Debugger](#) documentation.

Alternative debugging techniques

You can also use performance counters to measure performance with external tools. Many of these offer insight beyond the Unity Frame Debugger or GPU Profiler.

Build a standalone player and run it through any of the following:

- [Visual Studio Graphics Diagnostics](#) is a suite of tools for analyzing Direct3D apps. You can capture the output of your application, then save the captured frames in its Graphics Analyzer. You can also get a rendering performance summary from its [Frame Analysis](#) tool.
- [Intel Graphics Performance Analyzers](#) is another graphics analyzer with support for more APIs (DirectX, Metal, Vulkan, and OpenGL). View CPU, GPU, and Graphics API metrics in real-time to determine whether your application is CPU- or GPU-bound. Capture frames with [Graphics Frame Analyzer](#), and pinpoint CPU and GPU activity with the [Graphics Trace Analyzer](#).

- [RenderDoc](#) is an open source frame debugger with support for Windows, Linux, and Android (Nintendo Switch™ support is distributed separately as part of the NintendoSDK). Unity supports [Editor integration](#) with RenderDoc to make frame debugging easier. Use RenderDoc, for example, to visualize overdraw with the URP.
- [NVIDIA Nsight Graphics](#) is a standalone developer tool that enables you to debug, profile, and export frames across a variety of graphics APIs (Direct3D, DirectX Raytracing, Vulkan, OpenGL, OpenVR, and the Oculus SDK).

Use the [Range Profiler](#) inside of Nsight Graphics to break down a set of serialized frames into GPU execution times. Analyze GPU throughput and utilization with minimal overhead using GPU Trace, or investigate GPU crashes and hangs with Nsight Aftermath.

- AMD offers their own [Radeon Developer Tool Suite](#), consisting of a [GPU Profiler](#) and [Analyzer](#). These add tools to analyze AMD GPUs with compatible graphics APIs (DirectX, Vulkan, OpenGL, and OpenCL).
- [Apple Xcode](#) includes [Instruments](#), which allows you to gather different types of data from your project and view them side by side. [Frame Capture](#) allows you to pause your app and take a snapshot of the Metal commands and buffers. You can then debug a visual frame graph that shows all of the render passes and their dependencies.

Optimize fill rate and reduce overdraw

Fill rate refers to the number of pixels the GPU can render to the screen each second.

If your game is limited by fill rate, this means that it's trying to draw more pixels per frame than the GPU can handle.

Drawing on top of the same pixel multiple times is called overdraw. Overdraw decreases fill rate and costs extra memory bandwidth. The most common causes of overdraw are:

- Overlapping opaque or transparent geometry
- Complex shaders, often with multiple render passes
- Unoptimized particles
- Overlapping UI elements

While you want to minimize its effect, there is no one-size-fits-all approach to solving overdraw problems. Begin by experimenting with the above factors to reduce their impact.

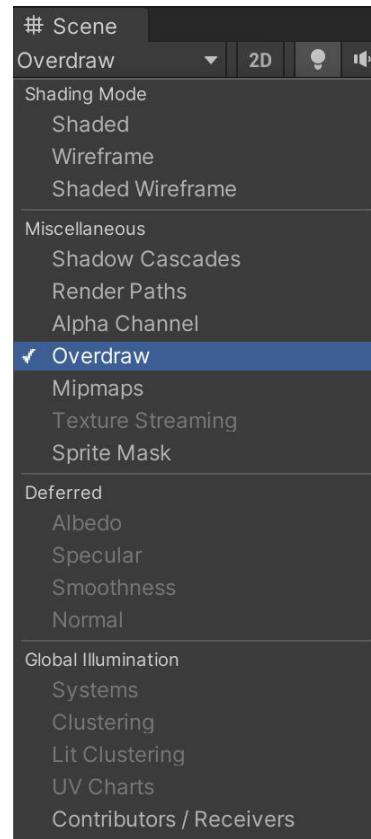
Draw order and render queues

To combat overdraw, you should understand how Unity sorts objects before rendering them.

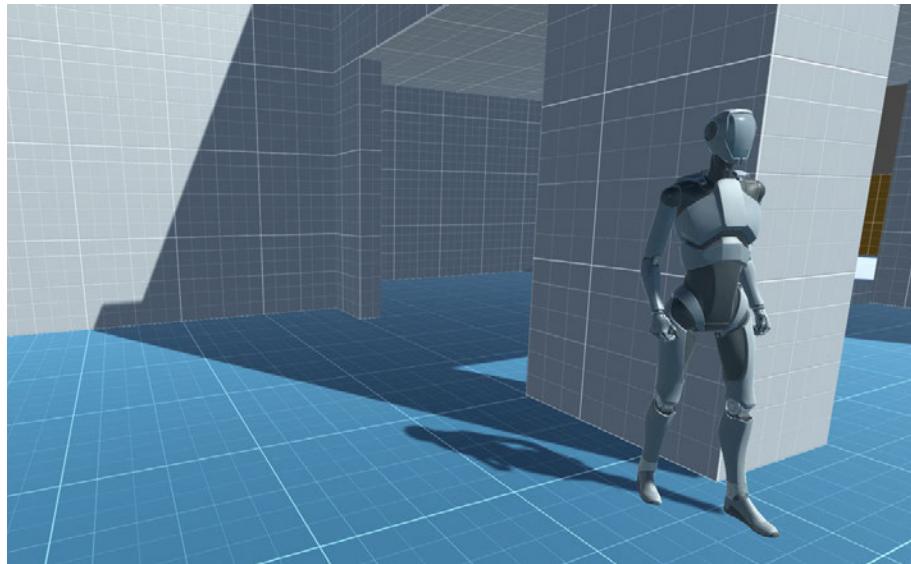
The Built-in Render Pipeline sorts GameObjects according to their [Rendering Mode](#) and [renderQueue](#). Each object's shader places it in a [render queue](#), which often determines its draw order.

Each render queue may follow different rules for sorting before Unity actually draws the objects to screen. For example, Unity sorts the Opaque Geometry queue front-to-back, but the Transparent queue sorts back-to-front.

Objects rendering on top of one another create overdraw. If you are using the Built-in render Pipeline, you can visualize overdraw in the [Scene view control bar](#). Switch the draw mode to Overdraw.

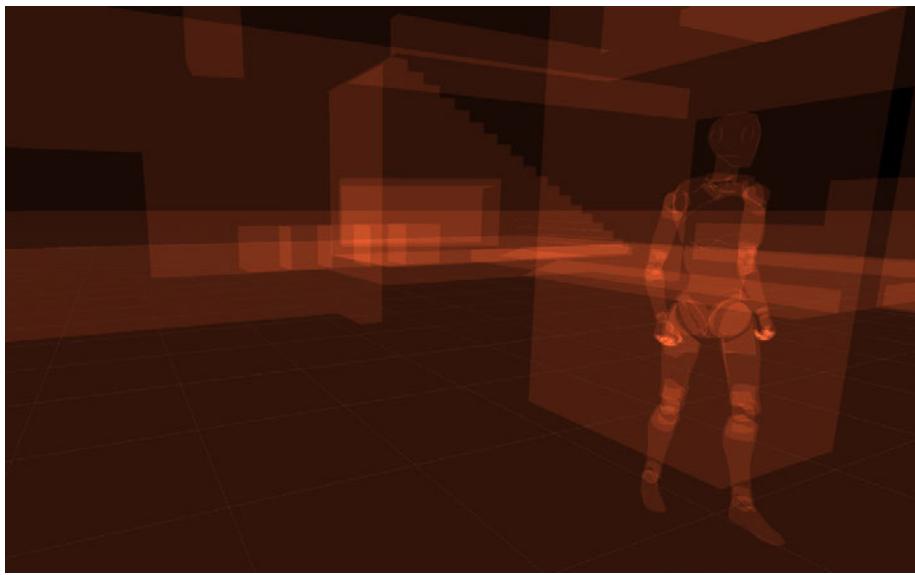


Overdraw in the Scene view control bar.



A Scene in standard Shaded view

Brighter pixels indicate objects drawing on top of one another; dark pixels mean less overdraw.



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

HDRP controls the render queue slightly differently. To calculate the order of the render queue, the HDRP:

- Groups meshes by shared Materials
- Calculates the rendering order of those groups based on [Material Priority](#)
- Sorts each group using each Mesh Renderer's Priority property.

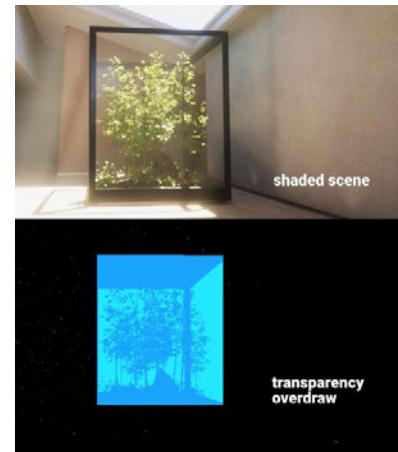
The resulting queue is a list of GameObjects that are first sorted by their Material's Priority, then by their individual Mesh Renderer's Priority.

This page on [Renderer and Material Priority](#) illustrates this in more detail.

To visualize transparency overdraw with HDRP, use the Render Pipeline Debug window (**Window > Render Pipeline > Render Pipeline Debug**) to select TransparencyOverdraw.

This debug option displays each pixel as a heat map going from black (which represents no transparent pixels) through blue to red (at which there are Max Pixel Cost number of transparent pixels).

When correcting overdraw, these diagnostic tools can offer a visual barometer of your optimizations.



The HDRP Render Pipeline Debug can visualize overdraw from transparent Materials.

Optimizing graphics for consoles

Though developing for Xbox and PlayStation does resemble working with their PC counterparts, those platforms do present their own challenges. Achieving smooth frame rates often means focusing on GPU optimization.

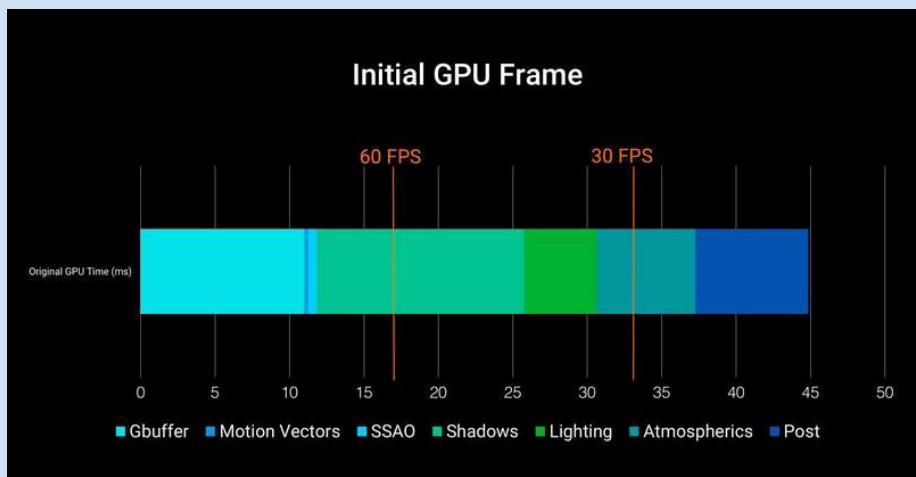


This section outlines the process of porting the *Book of the Dead* environment project to PlayStation 4.

Identify your performance bottlenecks

To begin, locate a frame with a high GPU load. Microsoft and Sony provide excellent tools for analyzing your project's performance on both the CPU and on the GPU. Make PIX for Xbox and Razor for PlayStation part of your toolbox when it comes to optimization on these platforms.

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



The view was GPU-bound on a PS4 Pro at roughly 45 milliseconds per frame.

Reduce the batch count

As with other platforms, optimization on console will often mean reducing draw call batches. There are a few techniques that might help.

- Use [Occlusion Culling](#) to 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](#) can also reduce your batches if you have many objects that share the same mesh and material. Limiting the 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](#) can reduce the GPU setup between DrawCalls by batching [Bind and Draw GPU commands](#). To benefit from this SRP batching, use as many Materials as needed, but restrict them to a small number of compatible shaders (e.g., Lit and Unlit Shaders in URP and HDRP).

Activate Graphics Jobs

Enable this option in **Player Settings > Other Settings** to take advantage of the PlayStation's or Xbox's multi-core processors. **Graphics Jobs (Experimental)** allows Unity to spread the rendering work across multiple CPU cores, removing pressure from the render thread. See [Multithreaded Rendering and Graphics Jobs tutorial](#) for details.

Profile the post-processing

Be sure to use post-processing assets that are optimized for consoles. Tools from the Asset Store that were originally authored for PC may consume more resources than necessary on Xbox or PlayStation. Profile using native profilers to be certain.

Avoid tessellation shaders

Tessellation subdivides shapes into smaller versions of that shape. This can enhance detail through increased geometry. Though there are examples where tessellation does make sense (e.g., [Book of the Dead](#)'s realistic tree bark), in general, avoid tessellation on consoles. They can be expensive on the GPU.

Replace geometry shaders with compute shaders

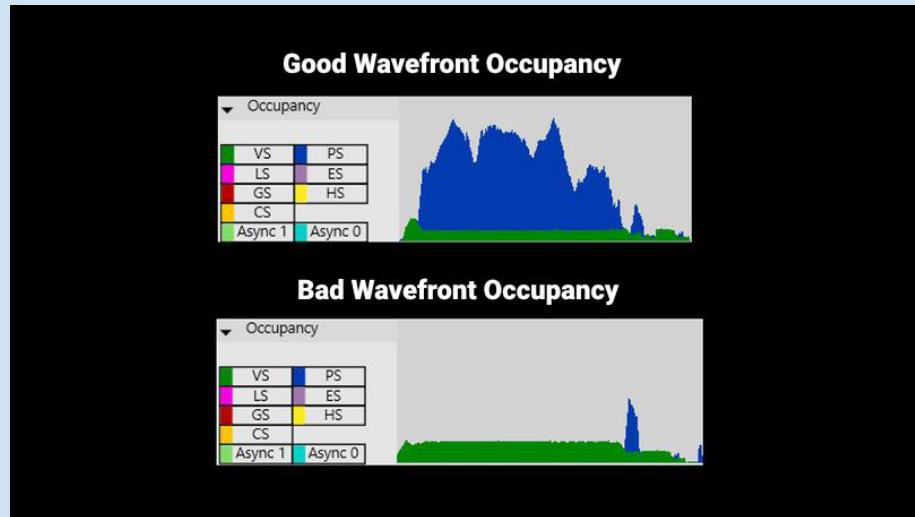
Like tessellation shaders, geometry and vertex shaders can run twice per frame on the GPU – once during the depth pre-pass, and again during the shadow pass.

If you want to generate or modify vertex data on the GPU, a [compute shader](#) is often a better choice than a geometry shader. Doing the work in a compute shader means that the vertex shader that actually renders the geometry can be comparatively fast and simple.

Aim for good waveform occupancy

When you send a draw call to the GPU, that work splits into many wavefronts that Unity distributes throughout the available SIMDs within the GPU.

Each SIMD has a maximum number of wavefronts that can be running at one time. *Wavefront occupancy* refers to how many wavefronts are currently in use relative to the maximum. This measures how well you are using the GPU's potential. PIX and Razor show waveform occupancy in great detail.



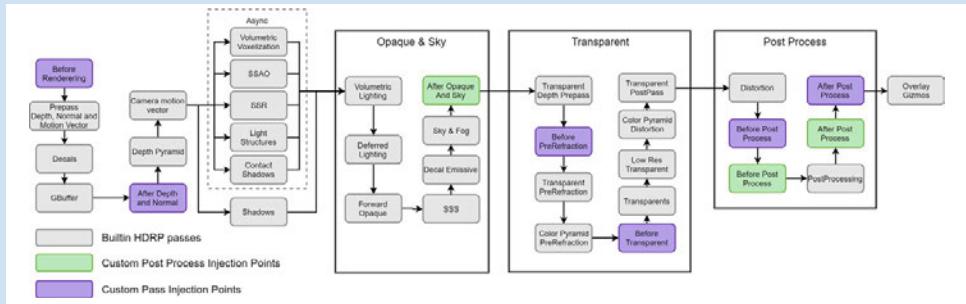
Good versus bad waveform occupancy

In this example from *Book of the Dead*, vertex shader wavefronts appear in green. Pixel shader wavefronts appear in blue. On the bottom graph, many vertex shader wavefronts appear without much pixel shader activity. This shows an underutilization of the GPU's potential.

If you're doing a lot of vertex shader work that doesn't result in pixels, that may indicate an inefficiency. While low waveform occupancy is not necessarily bad, it's a metric to start optimizing your shaders and checking for other bottlenecks. For example, if you have a stall due to memory or compute operations, increasing occupancy may help performance. On the other hand, too many in-flight wavefronts can cause cache thrashing and decrease performance.

Use HDRP built-in and custom passes

If your project uses HDRP, take advantage of its built-in and custom passes. These can assist in rendering the scene. The built-in passes can help you optimize your shaders. HDRP includes several injection points where you can add custom passes to your shaders.



Use HDRP injection points to customize the pipeline.

For optimizing the behavior of transparent materials, refer to this page on [Renderer and Material Priority](#).

Reduce the size of shadow mapping render targets

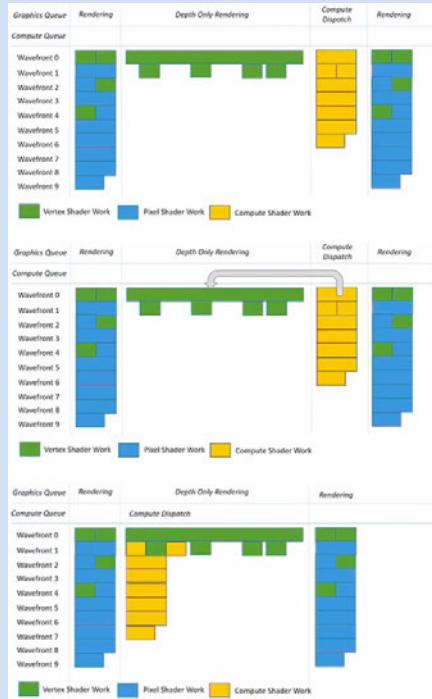
The High Quality setting of HDRP defaults to using a 4K shadow map. Reduce the shadow map resolution and measure the impact on the frame cost. Just be aware that you may need to compensate for any changes in visual quality with the light's settings.

Utilize Async Compute

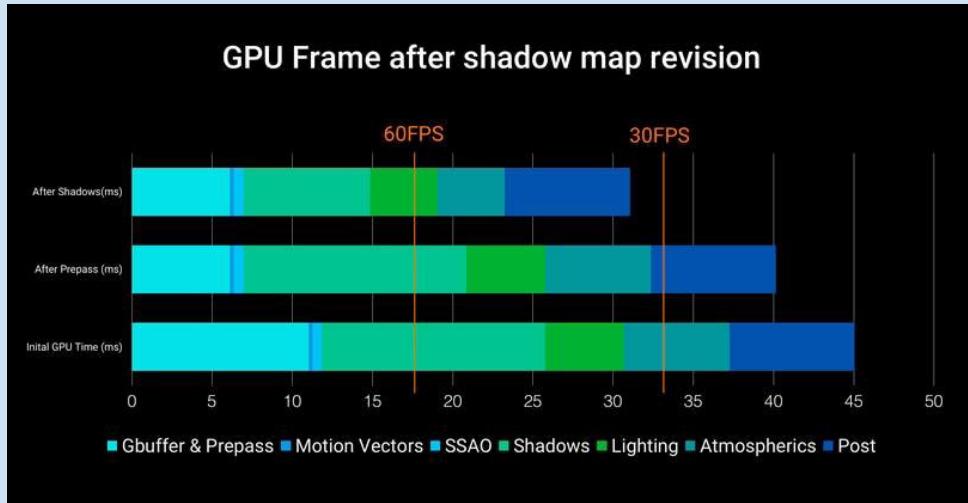
If you have intervals where you are underutilizing the GPU, Async Compute allows you to move useful compute shader work in parallel to your graphics queue. This makes better use of those GPU resources.

For example, during shadow map generation, the GPU performs depth-only rendering. Very little pixel shader work happens at this point, and many wavefronts remain unoccupied.

If you can synchronize some compute shader work with the depth-only rendering, this makes for a better overall use of the GPU. The unused wavefronts could help with Screen Space Ambient Occlusion or any task that complements the current work.



Async Compute can move compute shader work in parallel to the graphics queue.



Optimized render at 30 fps

In this example from *Book of the Dead*, several optimizations shaved several milliseconds off the shadow mapping, lighting pass, and atmospherics. The resulting frame cost allowed the application to run at 30 fps on a PS4 Pro.

Watch a performance case study in [Optimizing Performance for High-End Consoles](#), where Unity graphics developer Rob Thompson discusses porting the Book of the Dead to PlayStation 4. You can also read the corresponding [10 Tips for Optimizing Console Game Graphics](#) for more information.

Culling

[Culling](#) happens per camera. It can have a large impact on performance, especially when multiple cameras are enabled concurrently. Unity uses two types of culling:

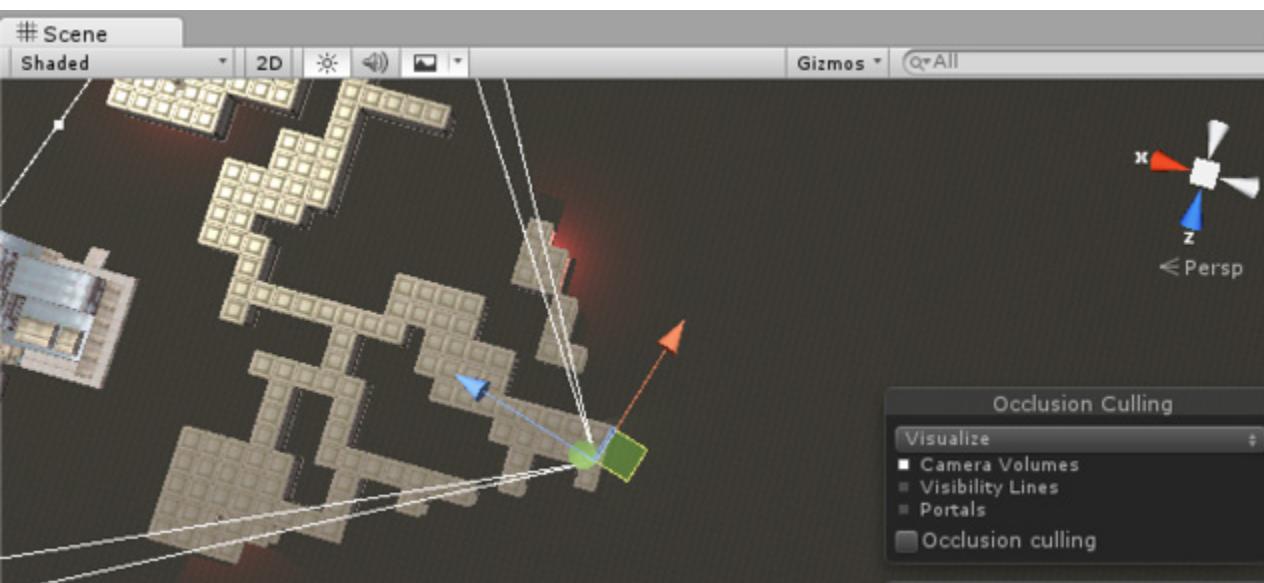
- **Frustum Culling** is performed automatically on every Camera. Frustum culling makes sure that GameObjects outside of the [View Frustum](#) are not rendered to save rendering performance.

You can set [per-layer culling distances](#) manually via [Camera.layerCullDistances](#). This allows you to cull small GameObjects at a distance shorter than the default [farClipPlane](#).

Organize GameObjects into Layers. Use the layerCullDistances array to assign each of the 32 layers a value less than the farClipPlane (or use 0 to default to the farClipPlane).

Unity culls by layer first. It only keeps any GameObjects on layers the Camera uses. Afterwards, frustum culling removes any GameObjects outside the camera frustum. Frustum culling is performed as a series of jobs to take advantage of available worker threads.

Each layer culling test is very fast (essentially just a bit mask operation). However, this cost could still add up with a very large number of GameObjects. If this becomes a problem for your project, you may need to implement some system to divide your world into “sectors” and disable sectors that are outside the Camera frustum in order to relieve some of the pressure on Unity’s layer/frustum culling system.



An example of
frustum culling

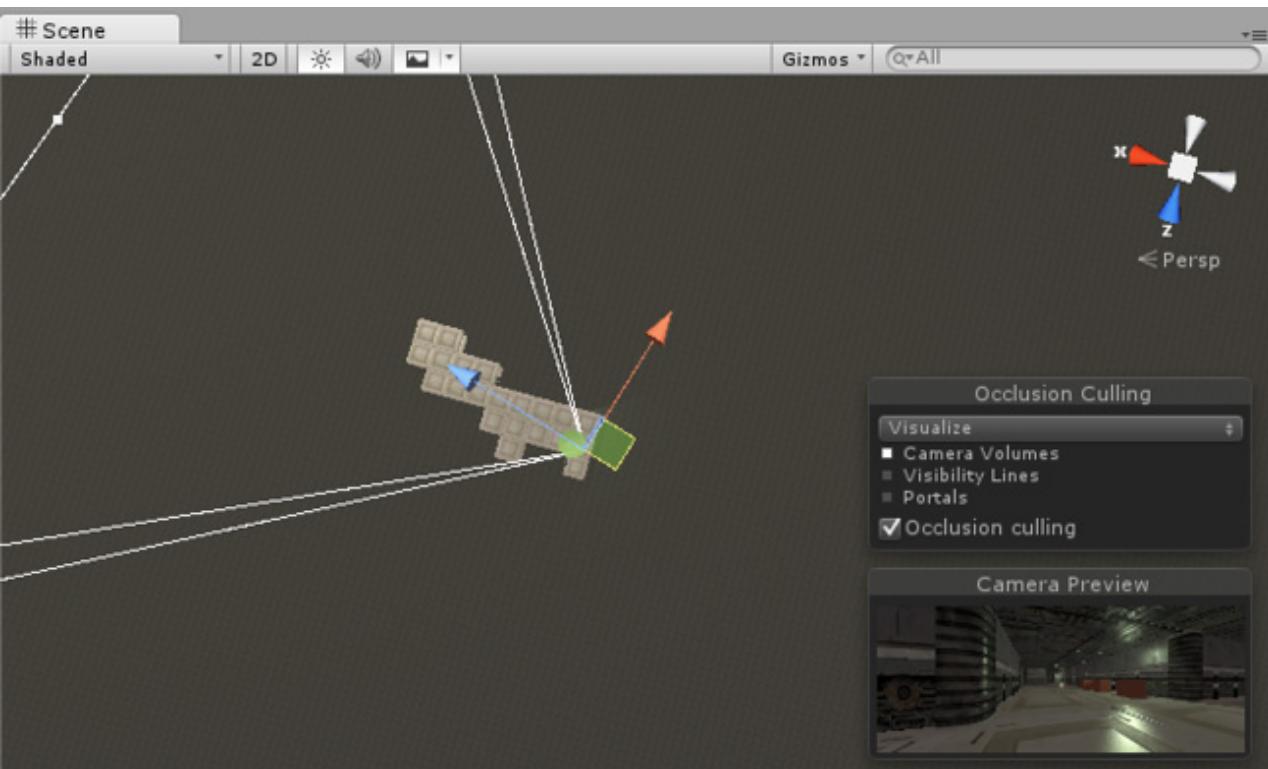
- [**Occlusion culling**](#) removes any **GameObjects** from the Game View if the Camera cannot see them. Objects hidden behind other objects can potentially still render and cost resources. Use occlusion culling to discard them.

For example, rendering another room is unnecessary if a door is closed and the Camera cannot see into the room.

If you enable occlusion culling, it may significantly increase performance but can use more disk space, CPU time, and RAM. Unity bakes the occlusion data during the build. It then needs to load it from disk to RAM while loading a Scene.

While frustum culling outside the camera view is automatic, occlusion culling is a baked process. Simply mark your objects as `Static.Occluders` or `Occludees`, then bake through the **Window > Rendering > Occlusion Culling** dialog.

Check out the [Working with Occlusion Culling tutorial](#) for more information.



An example of
occlusion culling

Dynamic resolution

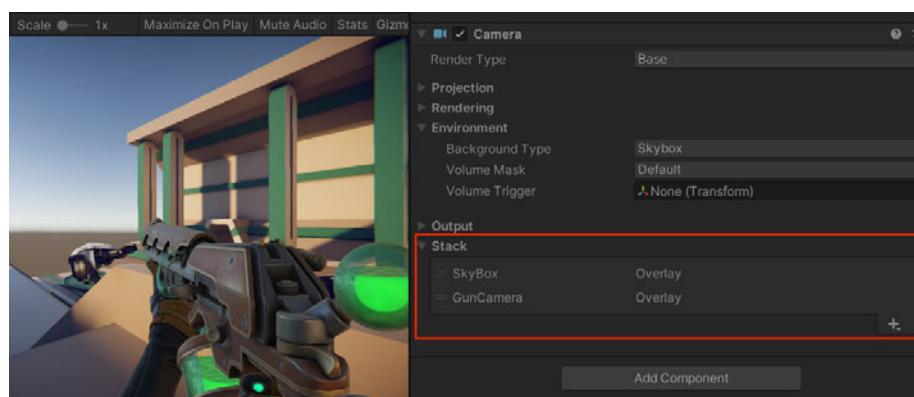
Allow Dynamic Resolution is a Camera setting that allows you to dynamically scale individual render targets to reduce workload on the GPU. In cases where the application's frame rate reduces, you can gradually scale down the resolution to maintain a consistent frame rate.

Unity triggers this scaling if performance data suggests that the frame rate is about to decrease as a result of being GPU-bound. You can also preemptively trigger this scaling manually with script. This is useful if you are approaching a GPU-intensive section of the application. If scaled gradually, dynamic resolution can be almost unnoticeable.

Refer to the [dynamic resolution](#) manual page for additional information and a list of supported platforms.

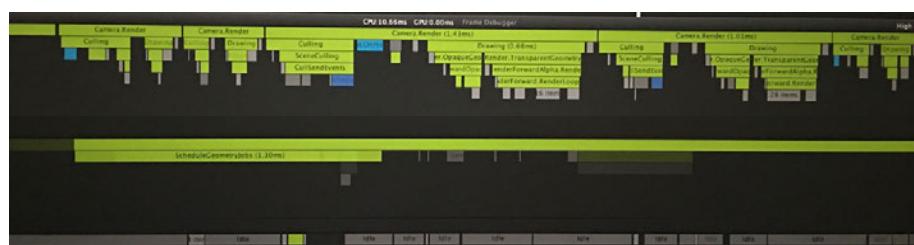
Multiple camera views

Sometimes you may need to render from more than one point of view during your game. For example, it's common in an FPS game to draw the player's weapon and the environment separately with different fields of view (FOV). This prevents the foreground objects from feeling too distorted viewed through the wide-angle FOV of the background.



Camera Stacking in URP. In this example, the gun and background render with different camera settings.

You could use [Camera Stacking](#) in URP to render more than one camera view. However, there is still significant culling and rendering done for each camera. Each camera incurs some overhead, whether it's doing meaningful work or not. Only use Camera components required for rendering. On mobile platforms, each active camera can use up to 1 ms of CPU time, even when rendering nothing.



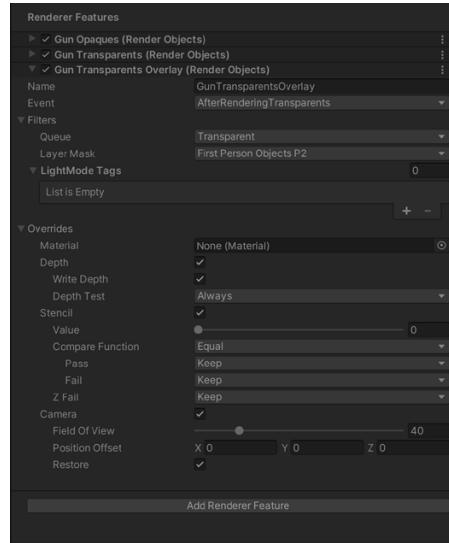
The Unity CPU Profiler shows the main thread in the timeline view and indicates that there are multiple Cameras. Unity performs culling for each Camera.

RenderObjects in URP

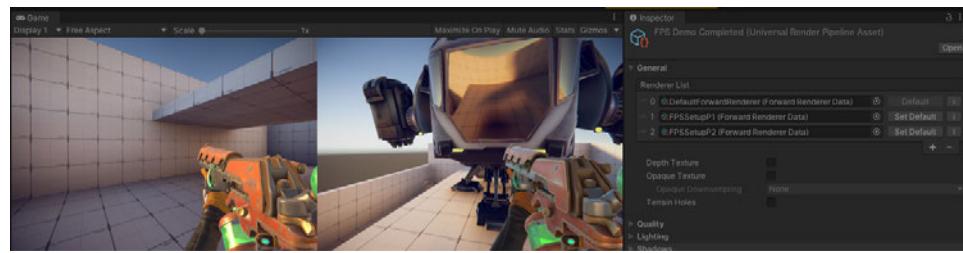
In the URP, instead of using multiple cameras, try a custom RenderObject. Select **Add Renderer Feature** in the **Renderer Data** asset. Choose **RenderObject (Experimental)**.

When overriding each RenderObject, you can:

- Associate it with an Event and inject it into a specific timing of the render loop
- Filter by Render Queue (Transparent or Opaque) and LayerMask
- Affect the Depth and Stencil settings
- Modify the Camera settings (Field of View and Position Offset)



Create a custom RenderObject to override render settings.



RenderObjects in URP combine multiple Layers into one rendered view.

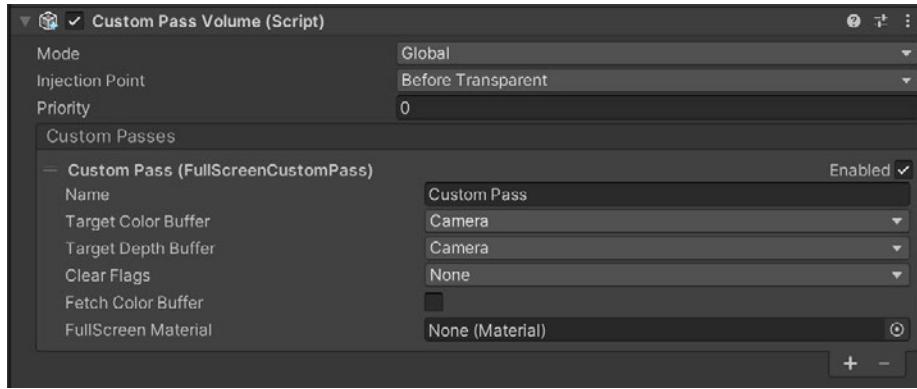
The [Universal Rendering Examples](#) project showcases this in action. Layer masks divide up the foreground and background meshes. Then, the custom Forward Renderer draws the weapon layer with a less distorted FOV and always renders it on top of the level geometry. Overrides on each Renderer modify the Stencil and Depth settings to prevent the foreground from clipping the rest of the scene geometry.

CustomPassVolumes in HDRP

In HDRP, you can use custom passes to similar effect. Configuring a [Custom Pass](#) using a Custom Pass Volume is analogous to using an [HDRP Volume](#).

A Custom Pass allows you to:

- Change the appearance of materials in your scene
- Change the order that Unity renders GameObjects
- Read Camera buffers into shaders



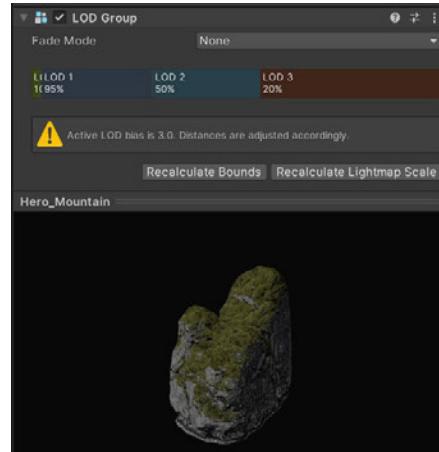
A CustomPassVolume in HDRP.

Using [Custom Passes in HDRP](#) can help you avoid using extra Cameras and the additional overhead associated with them. Custom passes have extra flexibility in how they can interact with shaders. You can also extend the [Custom Pass class with C#](#).

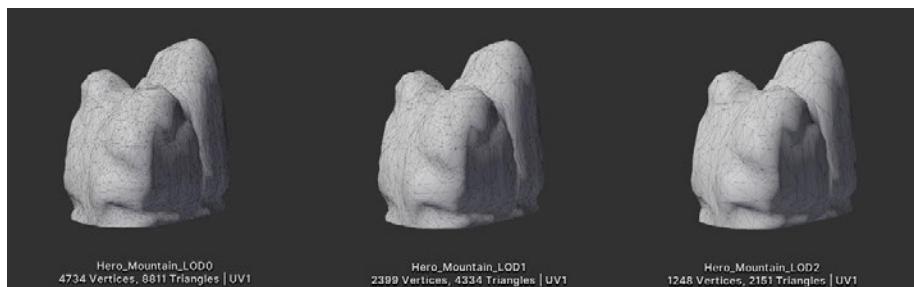
Use Level of Detail (LOD)

As objects move into the distance, [Level of Detail](#) can adjust or switch them to use lower-resolution meshes with simpler materials and shaders to aid GPU performance.

See the [Working with LODs](#) course on Unity Learn for more detail.



Example of a Mesh using a Level of Detail Group

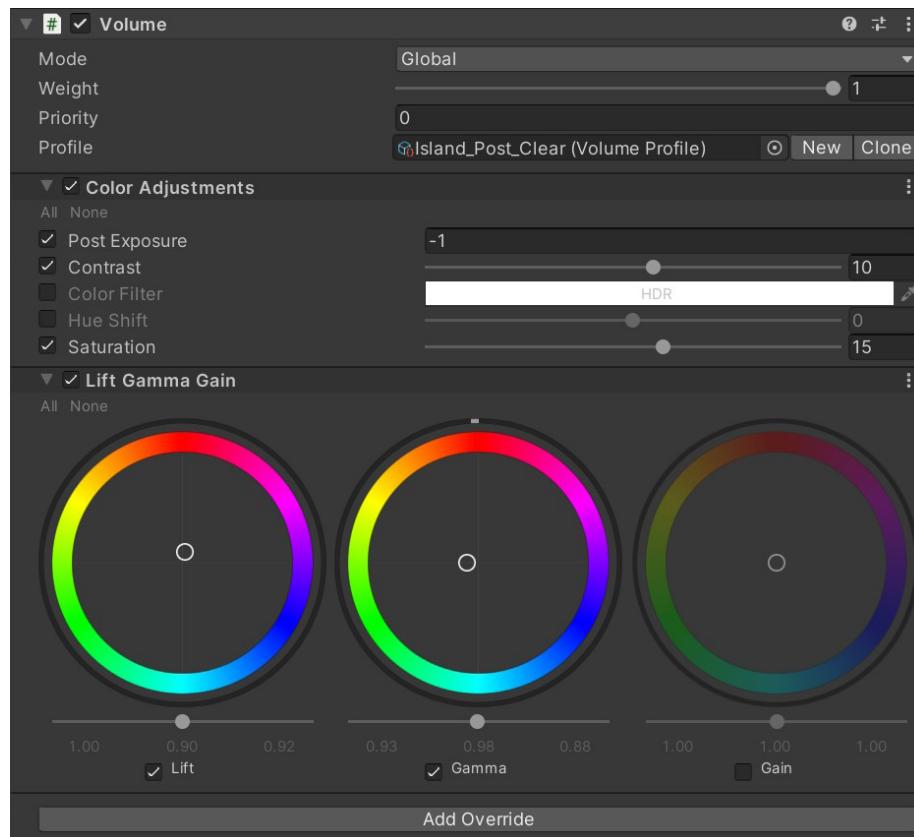


Source meshes, modeled at varying resolutions

Profile post-processing effects

Profile your [post-processing effects](#) to see their cost on the GPU. Some fullscreen effects, like Bloom and depth of field, can be expensive, but experiment until you find a happy balance between visual quality and performance.

Post-processing tends not to fluctuate much at runtime. Once you've determined your Volume Overrides, allot your post effects a static portion of your total frame budget.



Keep post-processing effects simple if possible.

User interface

[Unity UI \(UGUI\)](#) can often be a source of performance issues. The Canvas component generates and updates meshes for the UI elements and issues draw calls to the GPU. Its functioning can be expensive when misused, so keep the following factors in mind when working with UGUI.

Divide your Canvases

If you have one large Canvas with thousands of elements, updating a single UI element forces the whole Canvas to update, which can potentially generate a CPU spike.

Take advantage of UGUI's ability to support multiple Canvases. Divide UI elements based on how frequently they need to be refreshed. Keep static UI elements on a separate Canvas, and dynamic elements that update at the same time on smaller sub-canvases.

Ensure that all UI elements within each Canvas have the same Z value, materials, and textures.

Hide invisible UI elements

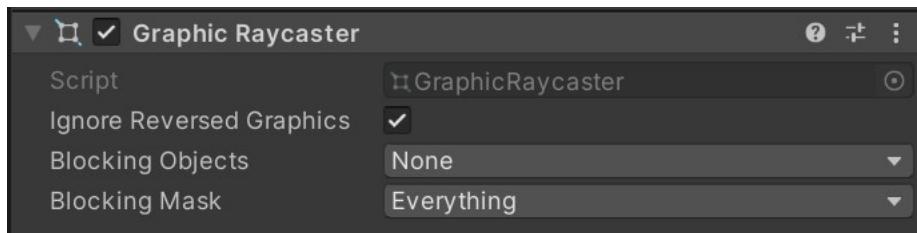
You might have UI elements that only appear sporadically in the game (e.g., a health bar that appears when a character takes damage). If your invisible UI element is active, it might still be using draw calls. Explicitly disable any invisible UI components and reenable them as needed.

If you only need to turn off the Canvas's visibility, disable the Canvas component rather than GameObject. This can save rebuilding meshes and vertices.

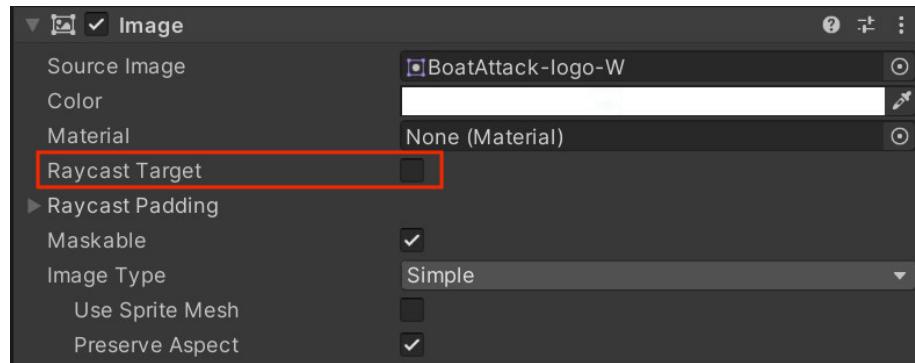
Limit GraphicRaycasters and disable Raycast Target

Input events like onscreen touches or clicks require the GraphicRaycaster component. This simply loops through each input point on screen and checks if it's within a UI's RectTransform. You need a Graphic Raycaster on every Canvas that requires input, including sub-canvases.

While this is not really a raycaster (despite the name), there is some expense for each intersection check. Minimize the number of Graphic Raycasters by not adding them to non-interactive UI Canvases.



In addition, disable Raycast Target on all UI text and images that don't need it. If the UI is complex with many elements, all of these small changes can reduce unnecessary computation.

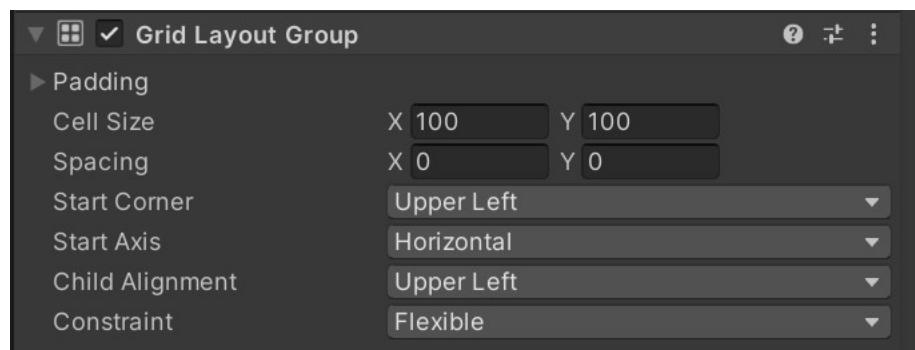


Disable Raycast Target if possible.

Avoid layout groups

Layout Groups update inefficiently, so use them sparingly. Avoid them entirely if your content isn't dynamic, and use anchors for proportional layouts instead. Otherwise, create custom code to disable the [Layout Group](#) components after they set up the UI.

If you do need to use Layout Groups (Horizontal, Vertical, Grid) for your dynamic elements, avoid nesting them to improve performance.



Layout Groups can lower performance, especially when nested.

Avoid large List and Grid views

Large List and Grid views are expensive. If you need to create a large List or Grid view (e.g., an inventory screen with hundreds of items), consider reusing a smaller pool of UI elements rather than creating a UI element for every item. Check out this sample [GitHub project](#) to see this in action.

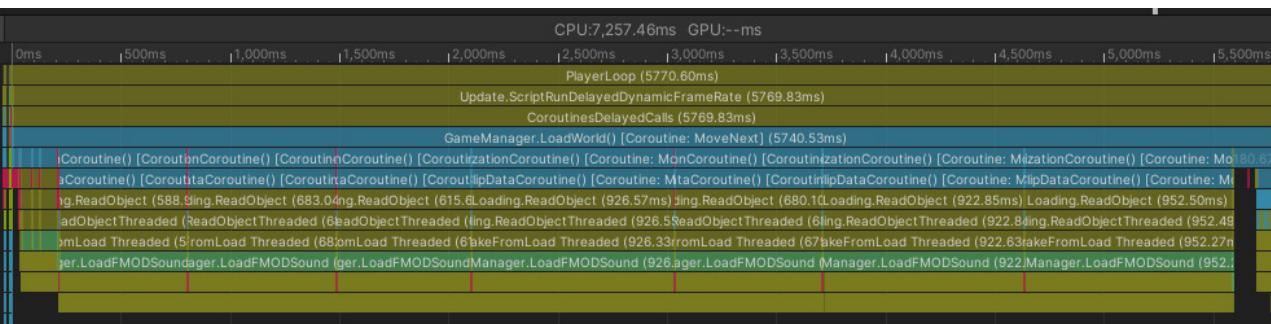
Avoid numerous overlaid elements

Layering lots of UI elements (e.g., cards stacked in a card battle game) creates overdraw. Customize your code to merge layered elements at runtime into fewer elements and batches.

When using a fullscreen UI, hide everything else

If your pause or start screen covers everything else in the scene, disable the camera that is rendering the 3D scene. Likewise, disable any background Canvas elements hidden behind the top Canvas.

Consider lowering the Application.targetFrameRate during a fullscreen UI, since you should not need to update at 60 fps.



Audio

Although audio is not normally a performance bottleneck, you can still optimize to save memory, disk space, or CPU usage.

Use lossless files as your source

Start with your sound assets in a lossless file format like WAV or AIFF.

If you use any compressed format (such as MP3 or Vorbis), then Unity will decompress it and recompress it during build time. This results in two lossy passes, degrading the final quality.

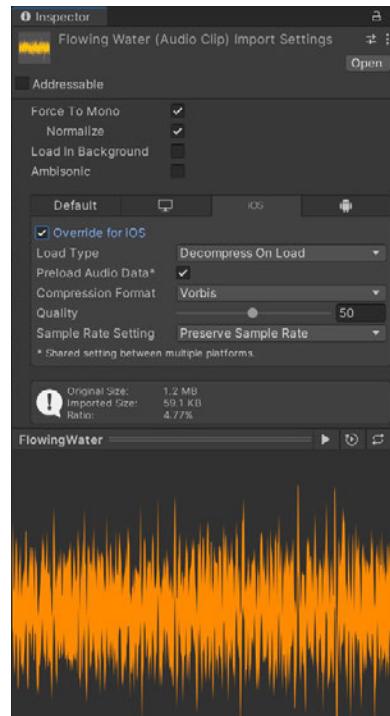


Reduce your AudioClips

Import settings on Audio Clips can save runtime memory and CPU performance:

- Enable the **Force To Mono** option on stereo audio files if they do not require stereo sound; this saves runtime memory and disk space.

Spatial Audio Sources should use AudioClips which are either authored in mono or have Force To Mono enabled in their import settings. If you use stereo sounds in spatial Audio Sources, the audio data will take up twice the disk space and memory; Unity must convert the sound to mono during the audio mixing process, which also requires extra CPU processing time.



AudioClip Import Settings

- **Preload Audio Data** ensures that Unity will load any referenced AudioClips before initializing the Scene. However, this may increase Scene loading times.
- If your sound clip is not needed immediately, load it asynchronously. Check **Load in Background**. This loads the sound at a delayed time on a separate thread, without blocking the main thread.
- Set the **Sample Rate Setting** to Optimize Sample Rate or Override Sample Rate.

For mobile platforms, 22050 Hz should be sufficient. Use 44100Hz (i.e., CD quality) sparingly. 48000Hz is excessive.

For PC/console platforms, 44100Hz is ideal. 48000Hz is usually unnecessary.

- Compress the AudioClip and reduce the compression bitrate.

For mobile platforms, use Vorbis for most sounds (or MP3 for sounds not intended to loop) . Use ADPCM for short, frequently used sounds (e.g., footsteps, gunshots).

For PC/Xbox, use Microsoft's XMA format instead of Vorbis or MP3. Microsoft recommends a compression ratio between 8:1 and 15:1.

For Playstation, use the ATRAC9 format. This has less CPU overhead than Vorbis or MP3.

- The proper Load Type depends on the length of the clip.

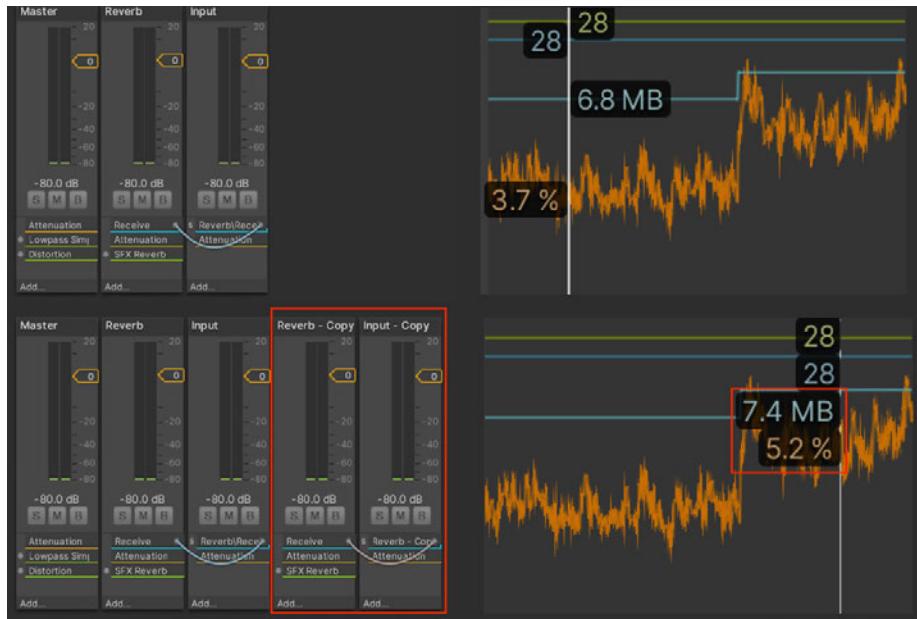
Clip size	Example usage	Load type settings
Small (< 200 KB)	Noisy sound effects (footsteps, gunshots), UI sounds	<p>Use Decompress on Load. This incurs a small CPU cost to decompress a sound into raw 16-bit PCM audio data, but will be the most performant at runtime.</p> <p>OR</p> <p>Set to Compressed In Memory and set Compression Format to ADPCM. This offers a fixed 3.5:1 compression ratio and is inexpensive to decompress in real-time.</p>
Medium (>= 200 KB)	Dialog, short music, medium/non-noisy sounds effects	<p>Optimal Load Type depends on the project's priorities.</p> <p>If reducing memory usage is the priority, select Compressed In Memory.</p> <p>If CPU usage is a concern, clips should be set to Decompress On Load.</p>
Large <td data-bbox="584 1127 915 1345">Background music, ambient background noise, long dialog</td> <td data-bbox="915 1127 1277 1345"> <p>Set to Streaming. Streaming has a 200 KB overhead, so it is only suitable for sufficiently large AudioClip.</p> </td>	Background music, ambient background noise, long dialog	<p>Set to Streaming. Streaming has a 200 KB overhead, so it is only suitable for sufficiently large AudioClip.</p>

Optimize the AudioMixer

In addition to your AudioClip settings, be aware of these issues with the AudioMixer.

- The [SFX Reverb Effect](#) is one of the most expensive audio effects in the AudioMixer. Adding a mixer group with SFX Reverb (and a mixer group sending to it) increases CPU cost.

This happens even if there is no AudioSource actually sending a signal to the groups. Unity's [Digital Signal Processing Graph \(DSPGraph\)](#) doesn't distinguish if it's getting null signals or not.



Adding a Reverb group and a group sending to it is expensive, even if no AudioSource writes to it.

- Reduce the number of mixer groups to improve AudioMixer performance.
Adding a large number of child groups under a single parent group increases audio CPU cost significantly. This happens even if all AudioSource outputs straight to Master, since Unity's DSP does not distinguish null signals.



An AudioMixer group with too many child groups

- Avoid parents with a single child group. Whenever possible, combine the two mixer groups into one.



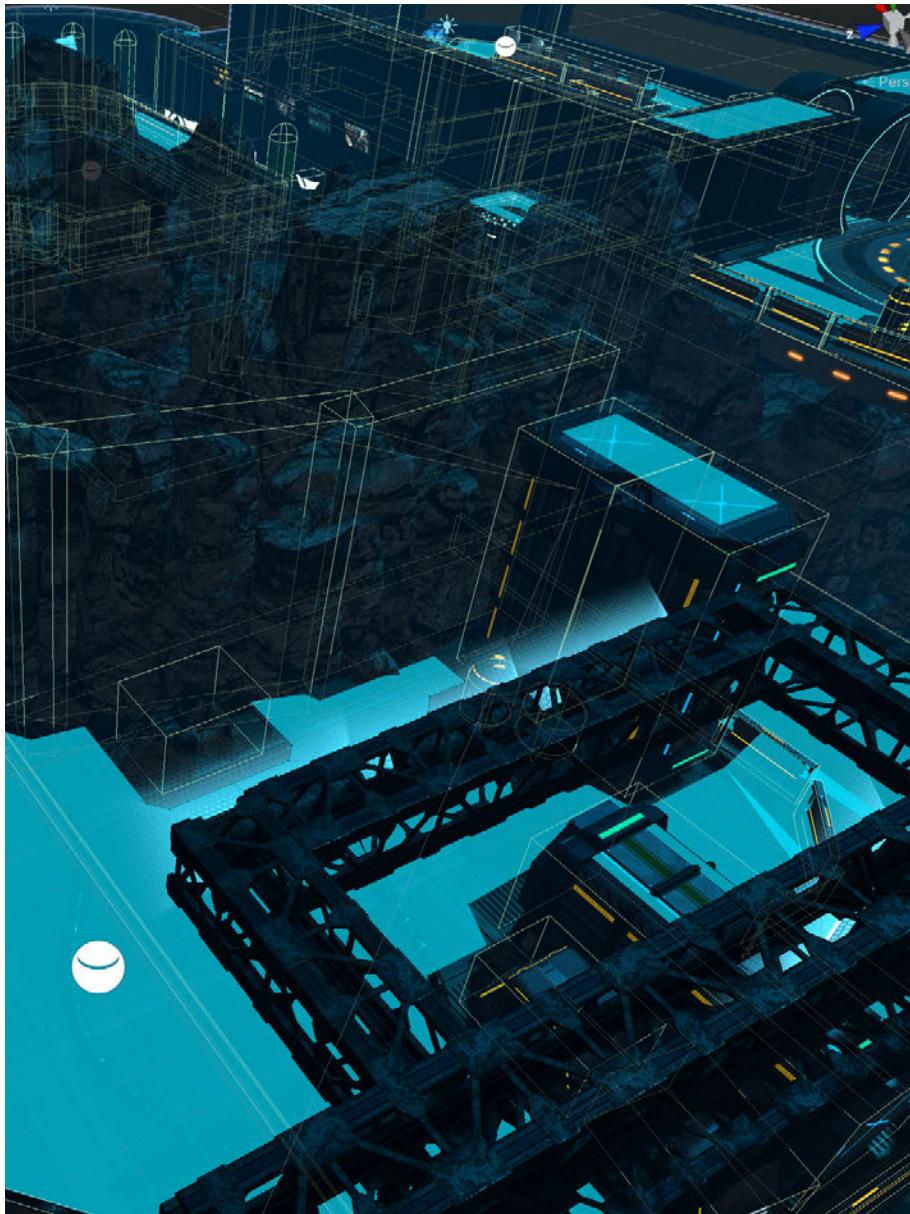
An AudioMixer with single child groups

Physics

Physics can create intricate gameplay, but this comes with a performance cost. When you know these costs, you can tweak the simulation to manage them appropriately. These tips can help you stay within your target frame rate and create smooth playback with Unity's built-in physics (NVIDIA PhysX).

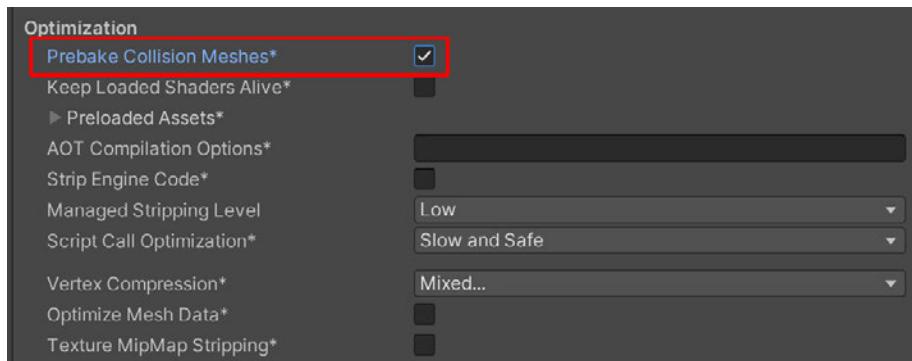
Simplify colliders

Mesh colliders can be expensive. Substitute more complex mesh colliders with primitive or simplified mesh colliders to approximate the original shape.



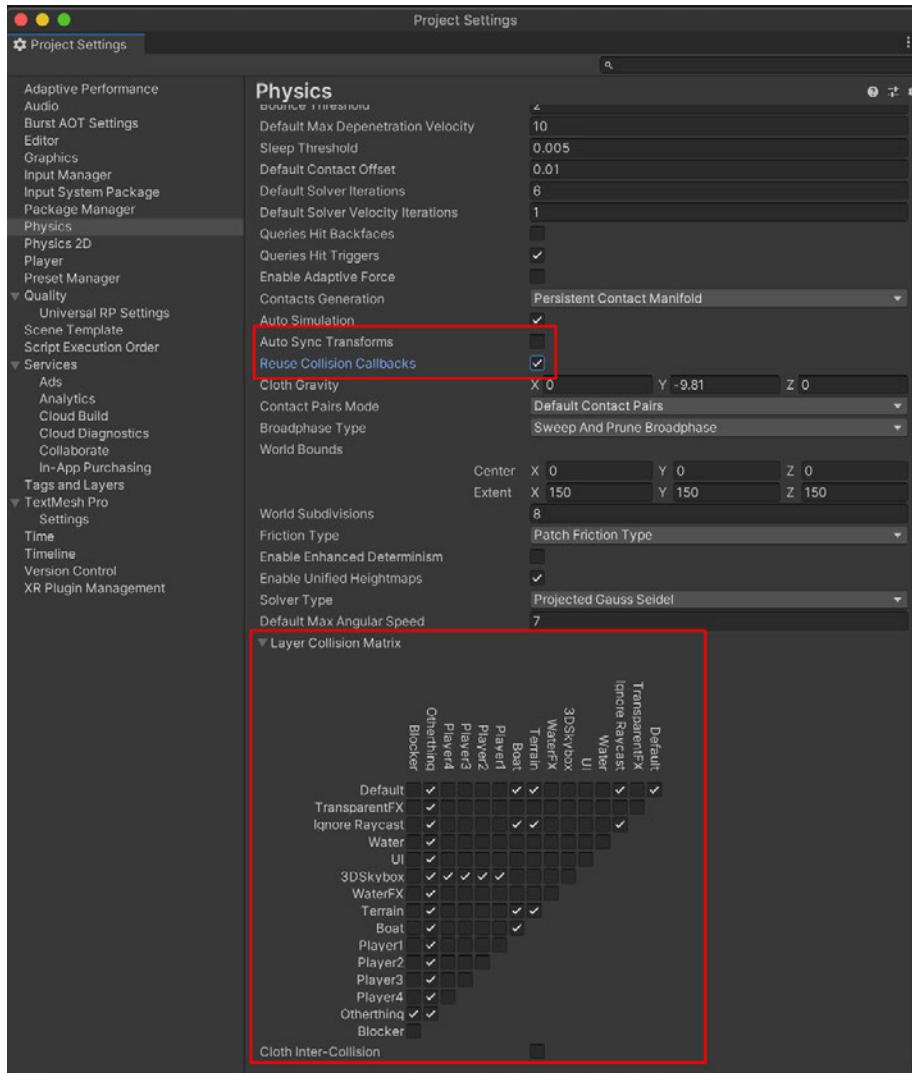
Optimize your settings

In the Player Settings, check **Prebake Collision Meshes** whenever possible.



Enable Prebake Collision Meshes

Make sure that you edit your Physics settings (**Project Settings > Physics**) as well. Simplify your Layer Collision Matrix wherever possible.

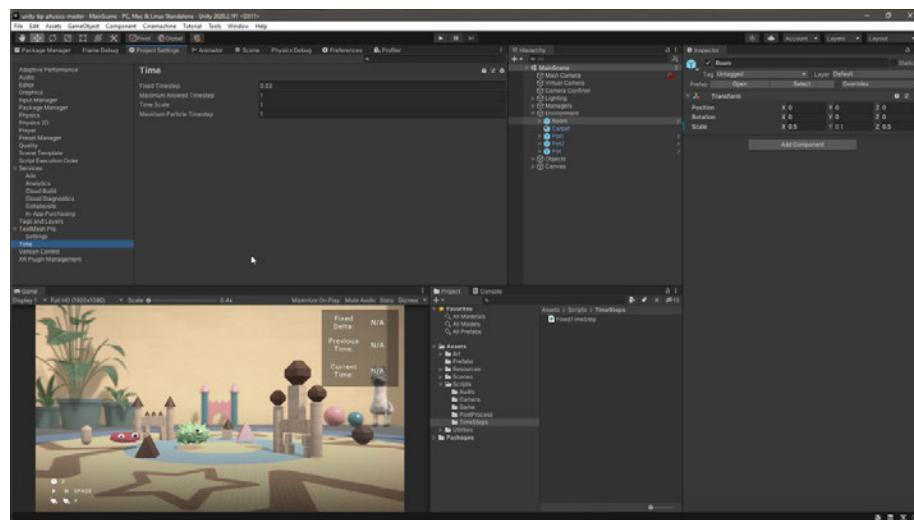


Modify the physics project settings to squeeze out more performance.

Adjust simulation frequency

Physics engines work by running on a fixed time step. To see the fixed rate that your project is running at, go to **Edit > Project Settings > Time**.

The **Fixed Timestep** field defines the time delta used by each physics step. For example, the default value of 0.02 seconds (20 ms) is equivalent to 50 fps, or 50 Hz.



The default Fixed Timestep in the Project Settings is 0.02 seconds (50 frames per second).

Because each frame in Unity takes a variable amount of time, it is not perfectly synced with the physics simulation. The engine counts up to the next physics time step. If a frame runs slightly slower or faster, Unity uses the elapsed time to know when to run the physics simulation at the proper time step.

In the event that a frame takes a long time to prepare, this can lead to performance issues. For example, if your game experiences a spike (e.g., instantiating many GameObjects or loading a file from disk), the frame could take 40 ms or more to run. With the default 20 ms Fixed Timestep, this would cause two physics simulations to run on the following frame in order to “catch up” with the variable time step.

Extra physics simulations, in turn, add more time to process the frame. On lower-end platforms, this potentially leads to a downward spiral of performance.

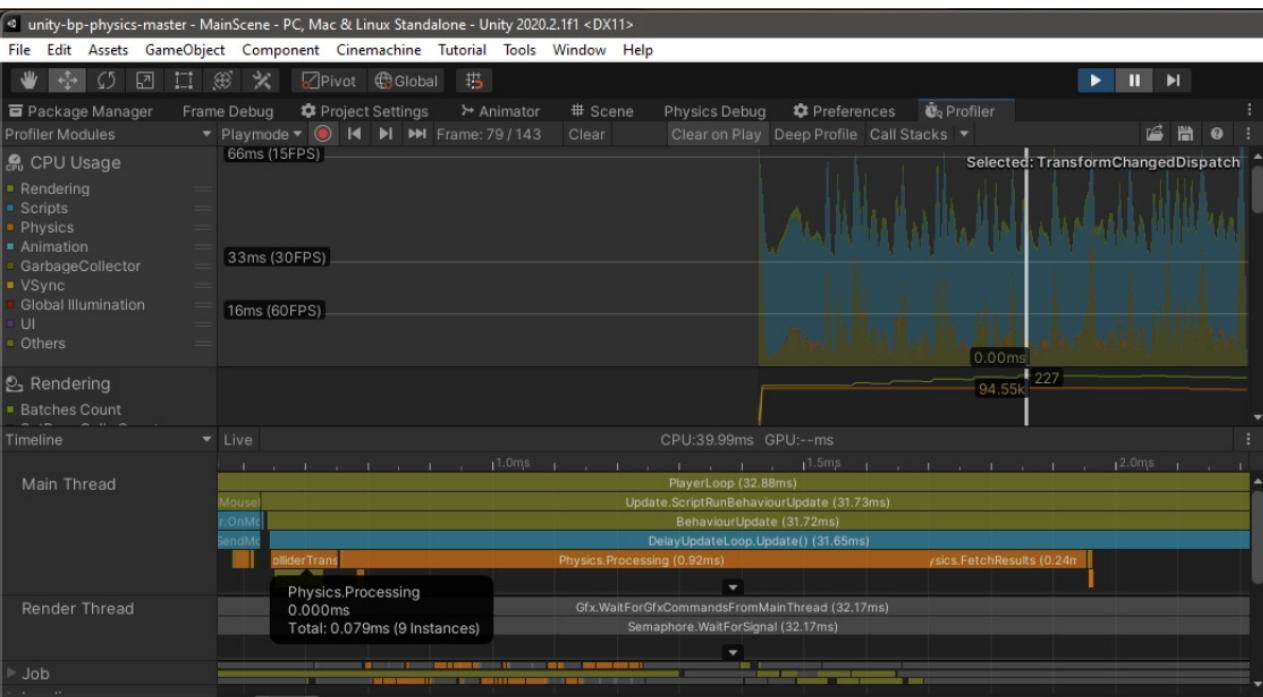
A subsequent frame taking longer to prepare makes the backlog of physics simulations longer as well. This leads to even slower frames and even more simulations to run per frame. The result is worse and worse performance.

Eventually the time between physics updates could exceed the Maximum Allowed Timestep. After this cutoff, Unity starts dropping physics updates, and the game stutters.

To avoid performance issues with physics:

- Reduce the simulation frequency. For lower-end platforms, increase the **Fixed Timestep** to slightly more than your target frame rate. For example, use 0.035 seconds for 30ps on mobile. This could help prevent that downward performance spiral.
- Decrease the **Maximum Allowed Timestep**. Using a smaller value (like 0.1 s) sacrifices some physics simulation accuracy, but also limits how many physics updates can happen in one frame. Experiment with values to find something that works for your project's requirements.
- Simulate the physics step manually if necessary. You can disable **Auto Simulation** in the Physics Settings and instead directly invoking [Physics.Simulate](#) during the Update phase of the frame. This allows you to take control when to run the physics step. Pass Time.deltaTime to Physics.Simulate in order to keep the physics in sync with the simulation time.

This approach can cause instabilities in the physics simulation in scenes with complex physics or highly variable frame times, so use it with caution.



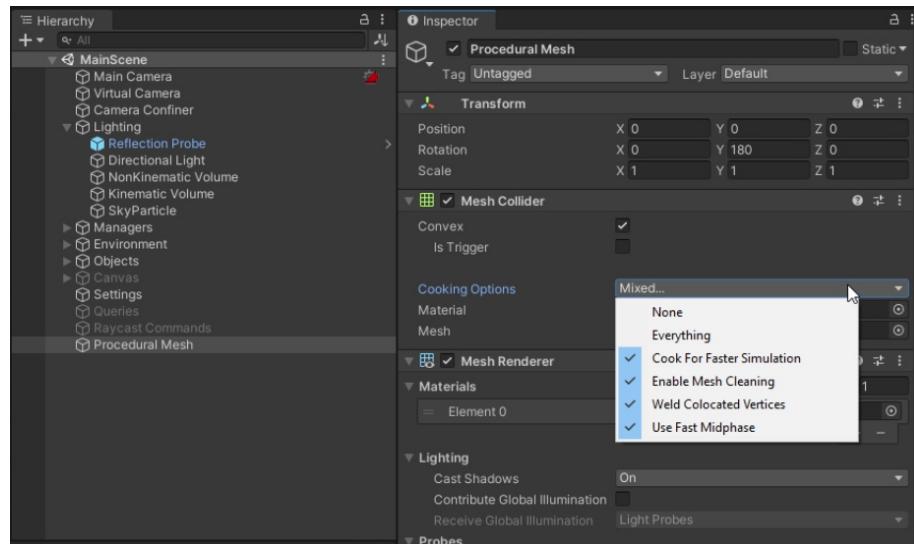
Modify CookingOptions for MeshColliders

Meshes used in physics go through a process called cooking. This prepares the mesh so that it can work with physics queries like raycasts, contacts, and so on.

A MeshCollider has several [CookingOptions](#) to help you validate the mesh for physics. If you are certain that your mesh does not need these checks, you can disable them to speed up your cook time.

In the CookingOptions for each MeshCollider, simply uncheck the `EnableMeshCleaning`, `WeldColocatedVertices`, and `CookForFasterSimulation`. These options are valuable for procedurally generated meshes at runtime, but can be disabled if your meshes already have the proper triangles.

Also, if you are targeting PC, make sure you keep `Use Fast Midphase` enabled. This switches to a faster algorithm from PhysX 4.1 during the mid-phase of the simulation (which helps narrow down a small set of potentially intersecting triangles for physics queries). Non-desktop platforms must still use the slower algorithm that generates [R-Trees](#).

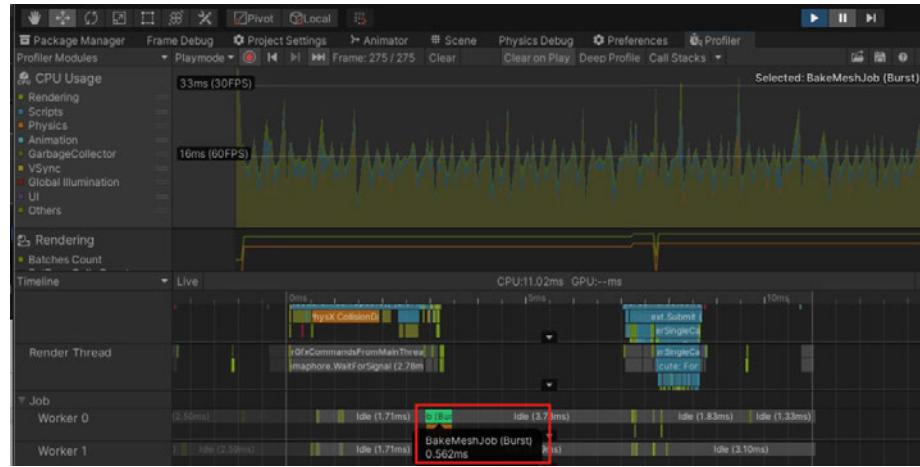


Use Physics.BakeMesh

If you are generating meshes procedurally during gameplay, you can create a Mesh Collider at runtime. Adding a MeshCollider component directly to the mesh, however, cooks/bakes the physics on the main thread. This can consume significant CPU time.

Use [Physics.BakeMesh](#) to prepare a mesh for use with a MeshCollider and save the baked data with the mesh itself. A new MeshCollider referencing this mesh will reuse this prebaked data (rather than baking the mesh again). This can help reduce Scene load time or instantiation time later.

To optimize performance, you can offload mesh cooking to another [thread](#) with the [C# Job System](#). Refer to [this example](#) for details on how to bake meshes across multiple threads.



Use Box Pruning for large scenes

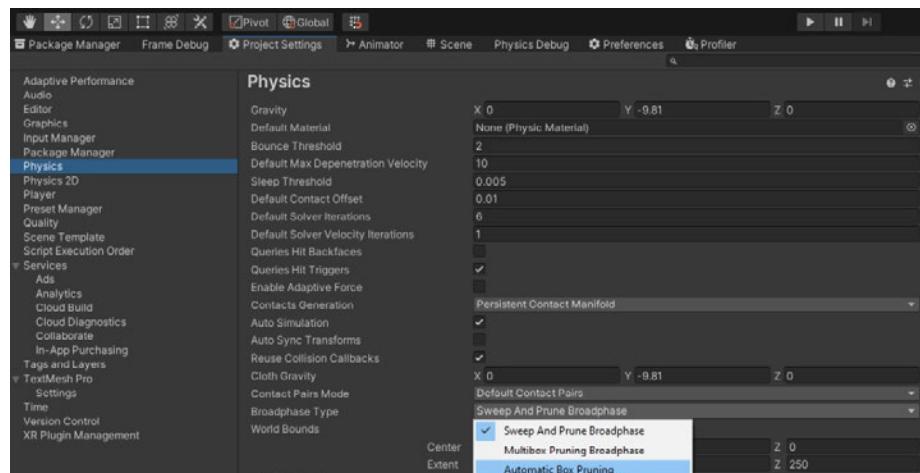
The Unity physics engine runs in two steps:

- the **broad phase**, which collects potential collisions using a [sweep and prune](#) algorithm
- the **narrow phase**, where the engine actually computes the collisions

The broad phase default setting of Sweep and Prune BroadPhase ([Edit > Project Settings > Physics > BroadPhase Type](#)) can generate false positives for worlds that are generally flat and have many colliders.

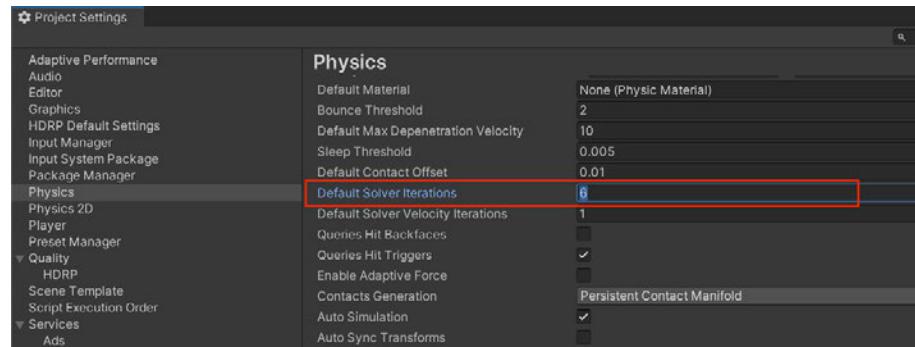
If your Scene is large and mostly flat, avoid this issue and switch to **Automatic Box Pruning** or **Multibox Pruning Broadphase**. These options divide the world into a grid, where each grid cell performs sweep-and-prune.

Multibox Pruning Broadphase allows you to specify the world boundaries and the number of grid cells manually, while Automatic Box Pruning calculates that for you.



Modify solver iterations

If you want to simulate a specific physics body more accurately, increase its [Rigidbody.solverIterations](#).



This overrides the `Physics.defaultSolverIterations`, which can also be found in [Edit > Project Settings > Physics > Default Solver Iterations](#).

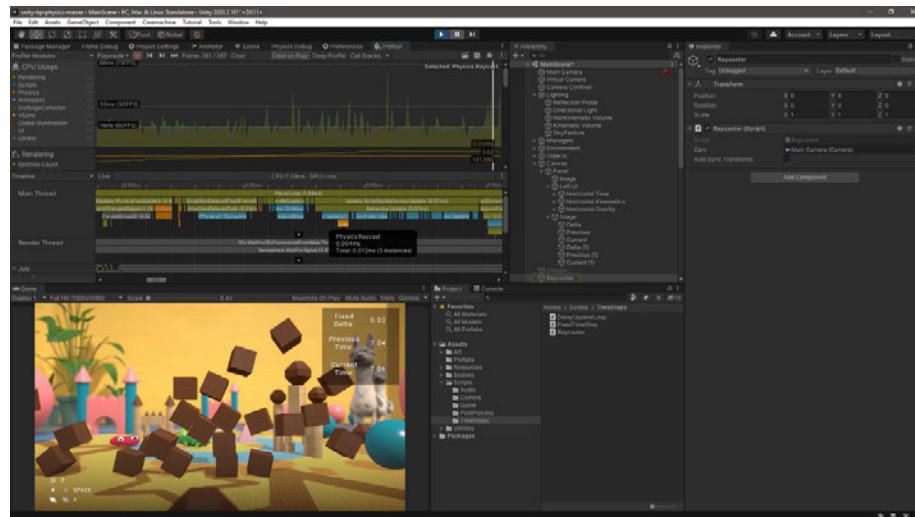
To optimize your physics simulations, set a relatively low value in the project's `defaultSolvlerIterations`. Then apply higher custom `Rigidbody.solverIterations` values to the individual instances that need more detail.

Disable automatic transform syncing

When you update a Transform, Unity does not automatically sync it to the physics engine. Unity accumulates transformations and waits for either the physics update to execute or for the user to call [Physics.SyncTransforms](#).

If you want to sync physics with your Transforms more frequently, you can set [Physics.autoSyncTransform](#) to true (also found in [Project Settings > Physics > Auto Sync Transforms](#)). When this is enabled, any [Rigidbody](#) or [Collider](#) on that [Transform](#) or its children automatically update with the Transform.

However, disable this unless absolutely necessary. Otherwise, a series of successive physics queries (such as raycasts) can lead to a loss in performance.



Profiling a Scene in Unity with Auto Sync Transform disabled

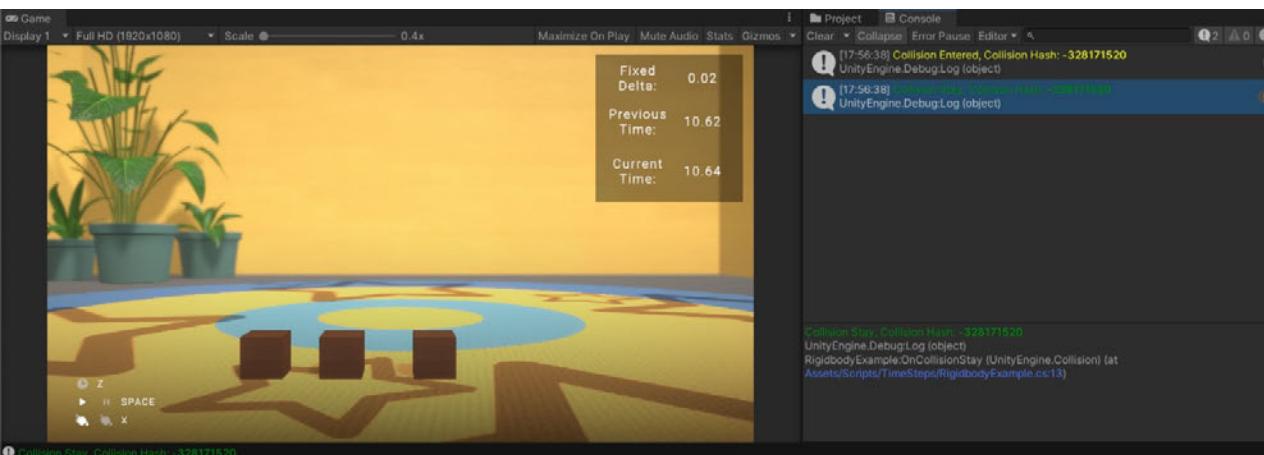
Reuse Collision Callbacks

The callbacks [MonoBehaviour.OnCollisionEnter](#), [MonoBehaviour.OnCollisionStay](#) and [MonoBehaviour.OnCollisionExit](#) all take a collision instance as a parameter. This collision instance is allocated on the managed heap and must be garbage collected.

To reduce the amount of garbage generated, enable [Physics.reuseCollisionCallbacks](#) (also found in **Projects Settings > Physics > Reuse Collision Callbacks**).

With this active, Unity only assigns a single collision pair instance to each callback. This reduces waste for the garbage collector and improves performance.

Note: If you reference the collision instance outside of the collision callbacks for post-processing, you must disable Reuse Collision Callbacks.



In the Unity Console, there is a single collision instance on Collision Entered and Collision Stay.

Move static colliders

Static colliders are GameObjects with a Collider component but without a Rigidbody.

Note that you can move a static collider, contrary to the term “static.” To do so, simply modify the position of the physics body. Accumulate the positional changes and sync before the physics update. You don’t need to add a Rigidbody component to the static collider just to move it.

However, if you want the static collider to interact with other physics bodies in a more complex way, give it a [kinematic Rigidbody](#). Use [Rigidbody.position](#) and [Rigidbody.rotation](#) to move it instead of accessing the Transform component. This guarantees more predictable behavior from the physics engine.

Note: In 2D physics, do not move static colliders because the tree rebuild is time consuming.

Use non-allocating queries

To detect and collect Colliders within a certain distance and in a certain direction, use raycasts and other physics queries like [BoxCast](#).

Physics queries that return multiple colliders as an array, like [OverlapSphere](#) or [OverlapBox](#), need to allocate those objects on the managed heap. This means that the garbage collector eventually needs to collect the allocated objects, which can decrease performance if it happens at the wrong time.

To reduce this overhead, use the **NonAlloc** versions of those queries. For example, if you are using OverlapSphere to collect all potential colliders around a point, use [OverlapSphereNonAlloc](#) instead.

This allows you to pass in an array of colliders (the results parameter) to act as a buffer. The NonAlloc method works without generating garbage. Otherwise, it functions like the corresponding allocating method.

Note that you need to define a results buffer of sufficient size when using a NonAlloc method. The buffer does not grow if it runs out of space.

Batch queries for ray casting

You can run raycast queries with [Physics.Raycast](#). However, if you have a large number of raycast operations (e.g., calculating line of sight for 10,000 agents), this may take a significant amount of CPU time.

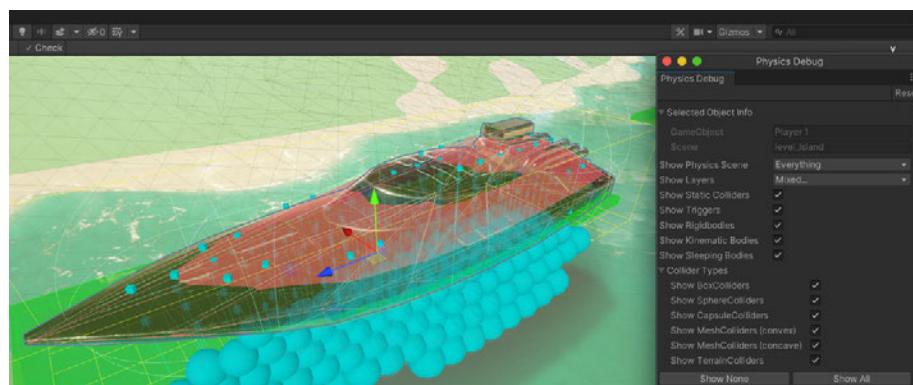
Use [RaycastCommand](#) to batch the query using the C# Job System. This offloads the work from the main thread so that the raycasts can happen asynchronously and in parallel.

See a usage example at the [RaycastCommands](#) documentation page.

Visualize with the Physics Debugger

Use the Physics Debug window (**Window > Analysis > Physics Debugger**) to help troubleshoot any problem colliders or discrepancies. This shows a color-coded indicator of the GameObjects that can collide with one another.

For more information, see [Physics Debug Visualization](#) in the Unity documentation.

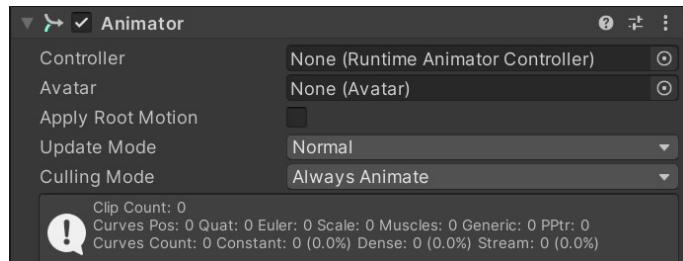


The Physics Debugger helps you visualize how your physics objects can interact with one another.

Animation

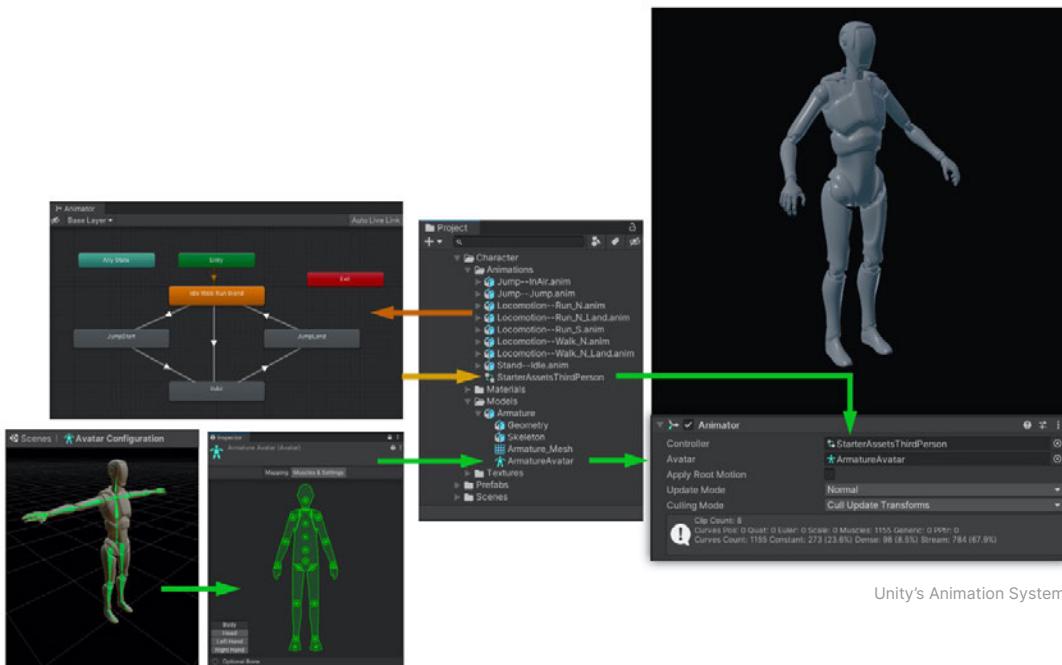
Unity's [Animation System](#) (sometimes called Mecanim) is fairly sophisticated. Its workflow involves several key components.

- [Animation Clips](#) contain information about how certain objects should change their position, rotation, or other properties over time.
- The [Animator Controller](#), a structured flowchart-like system, acts as a [State Machine](#). This tracks the clip currently being played, as well as when the animations should change or blend together.
- A humanoid rig gives you the ability to [retarget](#) bipedal animation from any source (e.g., motion capture, the Asset Store, or some other third-party animation library) to your own character model. Unity's [Avatar](#) system maps humanoid characters to a common internal format, making this possible.



Animator component

- A GameObject has an [Animator component](#) to connect these parts together. This component references an Animator Controller and an Avatar (if required). The Animator Controller, in turn, references the Animation Clips it uses.



Unity's Animation System

These guidelines will help you when working with animation in Unity.

Use generic rather than humanoid rigs

By default, Unity imports animated models with the generic rig, but developers often switch to the humanoid rig when animating a character. Be aware of these issues with rigs:

- Use a generic rig whenever possible. Humanoid rigs calculate inverse kinematics and animation retargeting each frame, even when not in use. Thus, they consume 30–50% more CPU time than their equivalent generic rigs.
- When importing humanoid animation, use an Avatar Mask to remove IK Goals or finger animation if you don't need them.
- With generic rigs, using root motion is more expensive than not using it. If your animations don't use root motion, do not specify a root bone.



Generic rigs use less CPU time than humanoid rigs.

Use alternatives for simple animation

Animators are primarily intended for humanoid characters. However, they are often repurposed to animate single values (e.g., the alpha channel of a UI element). Avoid overusing Animators, particularly in conjunction with UI elements, since they come with extra overhead.

For simple animation, use the [legacy animation system](#) when possible. Playing a single [Animation Clip](#) with no blending can make Unity slower with Animators than with the legacy animation system. The old system samples the curve and directly writes into the Transform.

The current animation system is optimized for animation blending and more complex setups. It has temporary buffers used for blending, and there is additional copying of the sampled curve and other data.

Also, if possible, consider not using the animation system at all. Create [easing functions](#) or use a third-party tweening library where possible (e.g., [DOTween](#)). These can achieve very natural-looking interpolation with mathematical expressions.

Avoid scale curves

Animating scale curves is more expensive than animating translation and rotation curves. To improve performance, avoid scale animations.

Note: This does not apply to constant curves (curves that have the same value for the length of the [animation clip](#)). Constant curves are optimized, and these are less expensive than normal curves.

Update only when visible

Set the animator's [Culling Mode](#) to Based on Renderers, and disable the [skinned mesh renderer's](#) Update When Offscreen property. This saves Unity from updating animations when the character is not visible.

Optimize workflow

Other optimizations are possible at the Scene level:

- Use hashes instead of strings to query the Animator.
- Implement a small AI Layer to control the Animator. You can make it provide simple callbacks for OnStateChange, OnTransitionBegin, and other events.
- Use State Tags to easily match your AI state machine to the Unity state machine.
- Use additional curves to simulate events.
- Use additional curves to mark up your animations, for example in conjunction with [target matching](#).

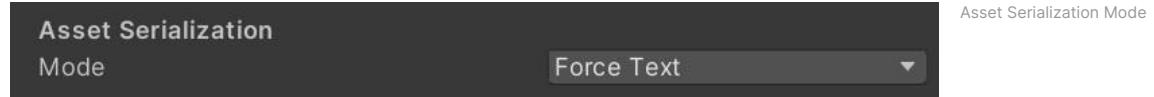
Workflow and collaboration

Building an application in Unity is a huge endeavor that often involves many developers. Make sure that your project is set up optimally for your team.

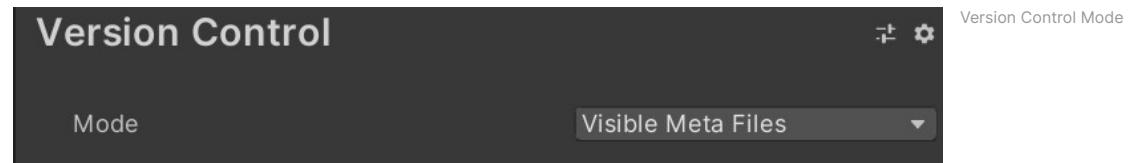
Use version control

Version control is essential for working as part of a team. It can help you track down bugs and bad revisions. Follow good practices like using branches and tags to manage milestones and releases.

To help with version control merges, make sure your Editor settings have **Asset Serialization Mode** set to **Force Text**. This is less space efficient but makes Unity store Scene files in a text-based format.



If you're using an external version control system (such as Git) in the Version Control settings, verify that the Mode is set to [Visible Meta Files](#).



Unity also has a built-in YAML (a human-readable, data-serialization language) tool specifically for merging Scenes and Prefabs. For more information, see [Smart merge](#) in the Unity documentation.

Plastic SCM

Most Unity projects include a sizable amount of art assets in addition to the script code. If you want to manage these assets with version control, consider switching to Plastic SCM. Even with Git LFS, Git does not perform as well as [Plastic SCM](#) with larger repositories.

[Plastic SCM](#) is our recommended version control solution for Unity game development. This solution offers a better experience when dealing with large binary files (>500 MB).



Plastic SCM allows you to:

- Work knowing that your art assets are securely backed up
- Track ownership of every asset
- Roll back to previous iterations of an asset
- Drive automated processes on a single central repository
- Create branches quickly and securely over multiple platforms

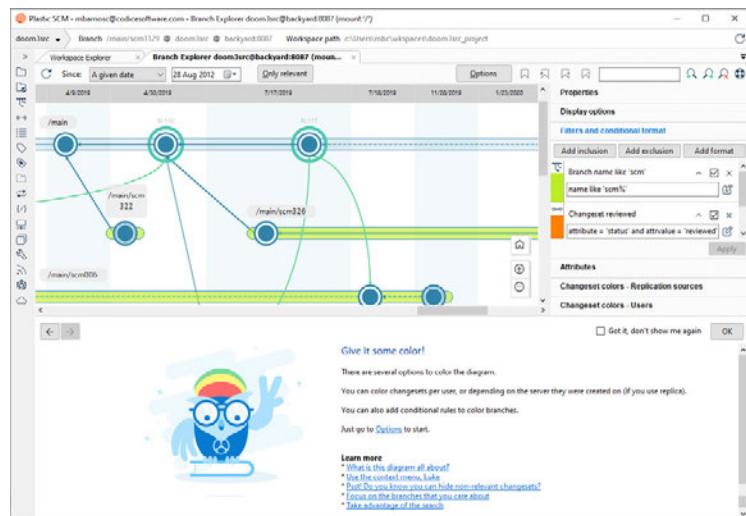
Plastic helps you centralize your development with excellent visualization tools. Artists especially will appreciate the [Gluon interface](#), which encourages tighter integration between development and art teams.

The [Plastic for Unity plugin](#) also lets you access this version control system (VCS) features straight from the Unity Editor.

To get started with Plastic SCM, create an account and download the [Quick installation guide](#).

Break up large Scenes

Large, single Unity Scenes do not lend themselves well to collaboration.



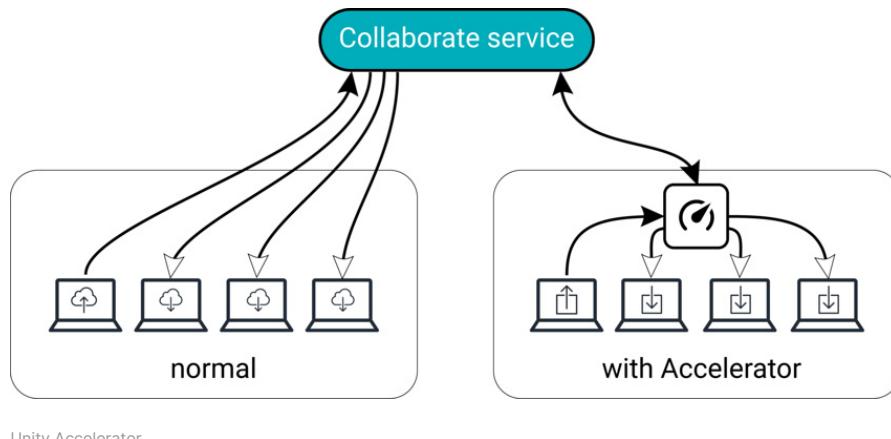
PlasticSCM offers visualization tools to help source control, even for large assets.

Break your levels into many smaller Scenes so that artists and designers can collaborate effectively on a single level while minimizing the risk of conflicts.

Note that at runtime, your project can load Scenes additively using **SceneManager.LoadSceneAsync** passing the **LoadSceneMode.Additive** parameter mode.

Speed up sharing with Unity Accelerator

The Unity Accelerator is a proxy and cache for the [Collaborate](#) service that allows you to share Unity Editor content faster. If your team is working on the same local network, you don't need to rebuild portions of your project, which significantly reduces download time. When used with [Unity Teams](#) Advanced, the Accelerator also shares source assets.



Unity Accelerator

Access the help you need with Unity Integrated Success

If you need personalized attention, consider [Unity Integrated Success](#). Integrated Success is much more than a support package. Integrated Success customers have the option to add read and modification access to Unity source code, among other benefits. This access is available for development teams that want to deep dive into Unity source code to adapt and reuse it for other applications.

[Unity Professional Services](#) conducts an in-depth analysis of your project's code and assets to identify areas for optimization. The teams deliver a report with actionable recommendations based on Unity best practices to help you maximize efficiency and performance.

Get a Project Review

Project Reviews are an essential part of the Integrated Success package. Whenever possible, the team travels directly to customers and typically spends two full days getting familiar with their projects before using profiling tools to detect performance bottlenecks. The Project Review also seeks to identify points where performance could be optimized for greater speed, stability, and efficiency.

For well-architected projects that have low build times (modular scenes, heavy usage of AssetBundles, etc.), changes are performed onsite, then the team reprofiles to uncover new issues.

In instances where problems can't be solved immediately, the review begins by capturing as much information as possible, then investigating further at Unity offices, in consultation with R&D developers as needed.

Though deliverables vary depending on customer needs, findings and recommendations are summarized in a written report to help studios identify potential blockers, assess risk, validate solutions, and ensure that they're following best practices moving forward.

Developer Relations Manager (DRM)

In addition to a Project Review, Unity Integrated Success also comes with a Developer Relations Manager (DRM) – a strategic Unity advisor who will quickly become an extension of the team to help projects get the most out of Unity. The DRM provides studios with the dedicated technical and operational expertise required to preempt issues and keep your projects running smoothly, right up to launch and beyond.

To learn more about our Integrated Success packages and Project Reviews, please reach out to Unity Sales or fill out this [form](#).

Next steps

You can find additional optimization tips, best practices, and news on the [Unity Blog](#) and [Unity community forums](#), as well as through [Unity Learn](#) and the **#unitytips** hashtag.

Performance optimization is a vast topic that requires careful attention. It is vital to understand how your target hardware operates, along with its limitations. In order to find an efficient solution that satisfies your design requirements, you will need to master Unity's classes and components, algorithms and data structures, and your platform's profiling tools.

Unity's team is always here to help you find the right tools and services for support throughout your game development journey, from concept to commercialization. If you're ready to get going, [you can access Unity Pro today](#) or [talk to one of our experts](#) to learn about all the ways we're ready to help you realize your vision.



unity.com