



# OPTIMIZE YOUR MOBILE GAME PERFORMANCE

# Contents

<b>Introduction .....</b>	<b>6</b>
<b>Profiling.....</b>	<b>8</b>
Profile early, often, and on the target device.....	8
Focus on optimizing the right areas .....	10
Understand how the Unity Profiler works .....	11
Use the Profile Analyzer .....	15
Work on a specific time budget per frame.....	16
Account for device temperature.....	17
Determine if you are GPU-bound or CPU-bound .....	17
Test on a min-spec device .....	17
<b>Memory.....</b>	<b>18</b>
Use the Memory Profiler .....	19
Reduce the impact of garbage collection (GC) .....	20
Time garbage collection whenever possible .....	20
Use the Incremental Garbage Collector to split the GC workload .....	21
<b>Adaptive Performance.....</b>	<b>22</b>
<b>Programming and code architecture .....</b>	<b>24</b>
Understand the Unity PlayerLoop.....	26
Minimize code that runs every frame.....	26
Avoid heavy logic in Start/Awake .....	27
Avoid empty Unity events .....	27
Remove Debug Log statements .....	27
Use hash values instead of string parameters .....	28
Choose the right data structure .....	28
Avoid adding components at runtime.....	28

Cache GameObjects and components.....	29
Use object pools.....	29
Use ScriptableObjects.....	31
<b>Project configuration .....</b>	<b>33</b>
Reduce or disable Accelerometer Frequency .....	33
Disable unnecessary Player or Quality settings.....	34
Disable unnecessary physics .....	34
Choose the right frame rate .....	34
Avoid large hierarchies.....	35
Transform once, not twice.....	35
Assume Vsync is enabled .....	35
<b>Assets .....</b>	<b>36</b>
Import textures correctly.....	36
Compress textures.....	39
Adjust mesh import settings .....	39
Check your polygon counts.....	41
Automate your import settings using the AssetPostprocessor .....	41
Unity DataTools .....	41
Use the Addressable Asset System .....	41
<b>Graphics and GPU optimization.....</b>	<b>43</b>
GPU optimization .....	46
Benchmark the GPU.....	46
Watch the rendering statistics .....	46
Use draw call batching .....	47
Use the Frame Debugger .....	49
Avoid too many dynamic lights.....	50
Disable shadows .....	50

Bake your lighting into Lightmaps.....	51
Use Light Layers.....	52
Use Light Probes for moving objects .....	52
Use Level of Detail (LOD).....	54
Use Occlusion Culling to remove hidden objects .....	55
Avoid mobile native resolution .....	55
Limit use of cameras .....	55
Keep shaders simple .....	56
Minimize overdraw and alpha blending .....	57
Limit post-processing effects .....	57
Be careful with Renderer.material.....	57
Optimize SkinnedMeshRenderers.....	58
Minimize Reflection Probes .....	58
System Metrics Mali.....	58
<b>User interface .....</b>	<b>59</b>
UGUI performance optimization tips .....	60
Divide your Canvases.....	60
Hide invisible UI elements .....	60
Limit GraphicRaycasters and disable Raycast Target.....	60
Avoid Layout Groups .....	61
Avoid large List and Grid views.....	61
Avoid numerous overlaid elements.....	62
Use multiple resolutions and aspect ratios .....	62
When using a fullscreen UI, hide everything else .....	62
Assign the Camera to World Space and Camera Space Canvases .	63
UI Toolkit performance optimization tips .....	63

<b>Audio .....</b>	<b>65</b>
Make sound clips mono when possible .....	65
Use original uncompressed WAV files as your source assets .....	67
Compress the clip and reduce the compression bitrate.....	67
Choose the proper Load Type.....	67
Unload muted AudioSources from memory.....	67
<b>Animation .....</b>	<b>68</b>
Use generic versus humanoid rigs .....	68
Avoid excessive use of Animators .....	69
<b>Physics .....</b>	<b>70</b>
Optimize your settings.....	70
Simplify colliders .....	72
Move a Rigidbody using physics methods.....	73
Fix the Fixed Timestep.....	73
Visualize with the Physics Debugger .....	73
<b>Workflow and collaboration .....</b>	<b>74</b>
Use version control .....	74
Break up large scenes .....	75
Remove unused resources .....	75
Accelerate sharing with Unity Accelerator.....	76
Remove roadblocks with Unity Integrated Success .....	77
<b>Conclusion and next steps .....</b>	<b>78</b>

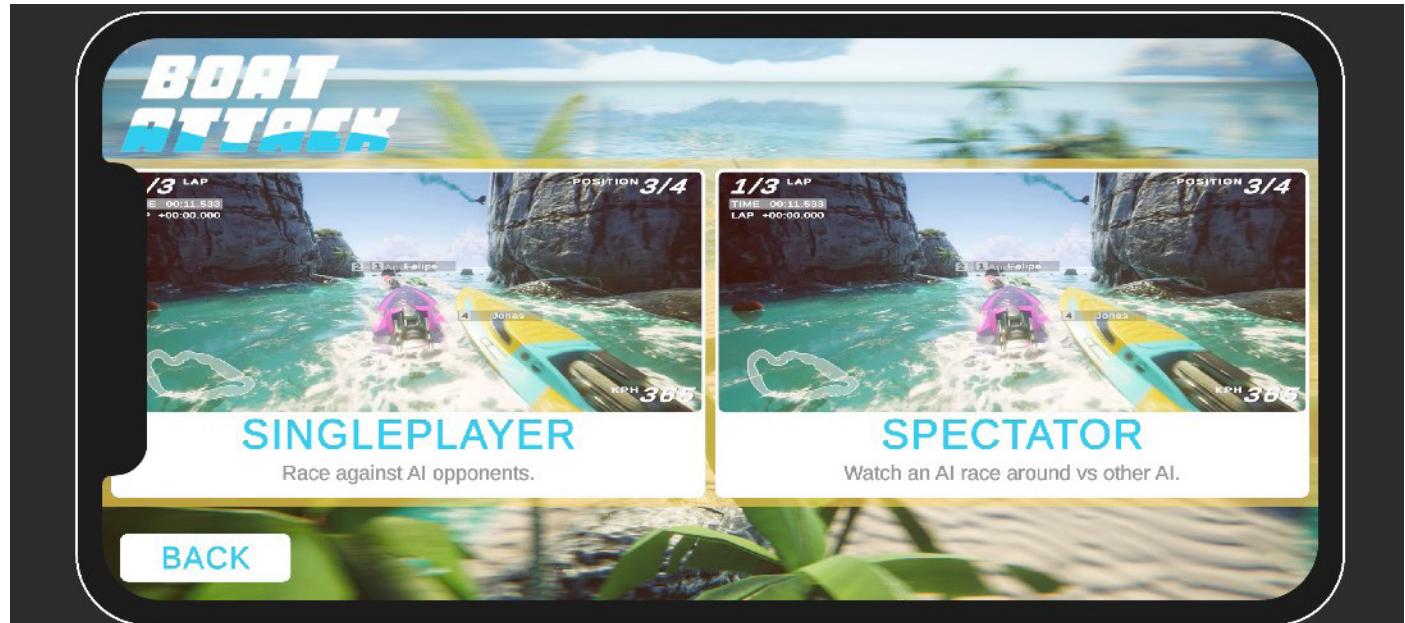
# INTRODUCTION

Optimizing your iOS and Android applications is an essential process that underpins the entire development cycle. Mobile hardware continues to evolve, and a mobile game's optimization – along with its art, game design, audio, and monetization strategy – plays a key role in shaping the player experience.

Both iOS and Android have active user bases in the *billions*. If your mobile game is highly optimized, it has a better chance at passing certification from platform-specific stores. To maximize your opportunity for success at launch and beyond, your aim is always twofold: building the slickest, most immersive experience and making it performant on the widest range of handhelds.

This guide assembles knowledge and advice from Unity's expert team of software engineers. [Unity's Accelerate Solutions](#) games team has partnered with developers across the industry to help launch the best games possible. Follow the steps outlined here to get the best performance from your mobile game while reducing its power consumption.

Start optimizing with support from the Unity team.<sup>1</sup>



<sup>1</sup>Note that many of the optimizations discussed here may introduce additional complexity, which can mean extra maintenance and potential bugs. Balance performance gains against the time and labor cost when implementing these best practices.

# PROFILING

## **Profile early, often, and on the target device**

Profiling is the process of measuring aspects of your game's performance at runtime. By using a profiling tool, you can measure how your game runs on its target platform and use this information to track down the cause of a performance problem. By watching the profiling tool as you make changes, you can gauge whether changes actually fix the performance problem.

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 and throughout the development cycle, not just when you are close to shipping. Investigate glitches or spikes as soon as they appear to benchmark performance before and after major changes in your project. 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.

Along with the Unity Profiler, you can leverage the [Memory Profiler](#) and [Profile Analyzer](#), as well as these native tools from iOS and Android for further performance testing on their respective hardware:

- On iOS, use [Xcode](#) and [Instruments](#).
- On Android / Arm use:

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

[Arm Mobile Studio](#): This suite of tools can help you profile and debug your games in great detail, catering toward devices running Arm hardware.

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

Certain hardware can also take advantage of [Intel VTune](#), which helps you to find and fix performance bottlenecks on Intel platforms (with Intel processors only).

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

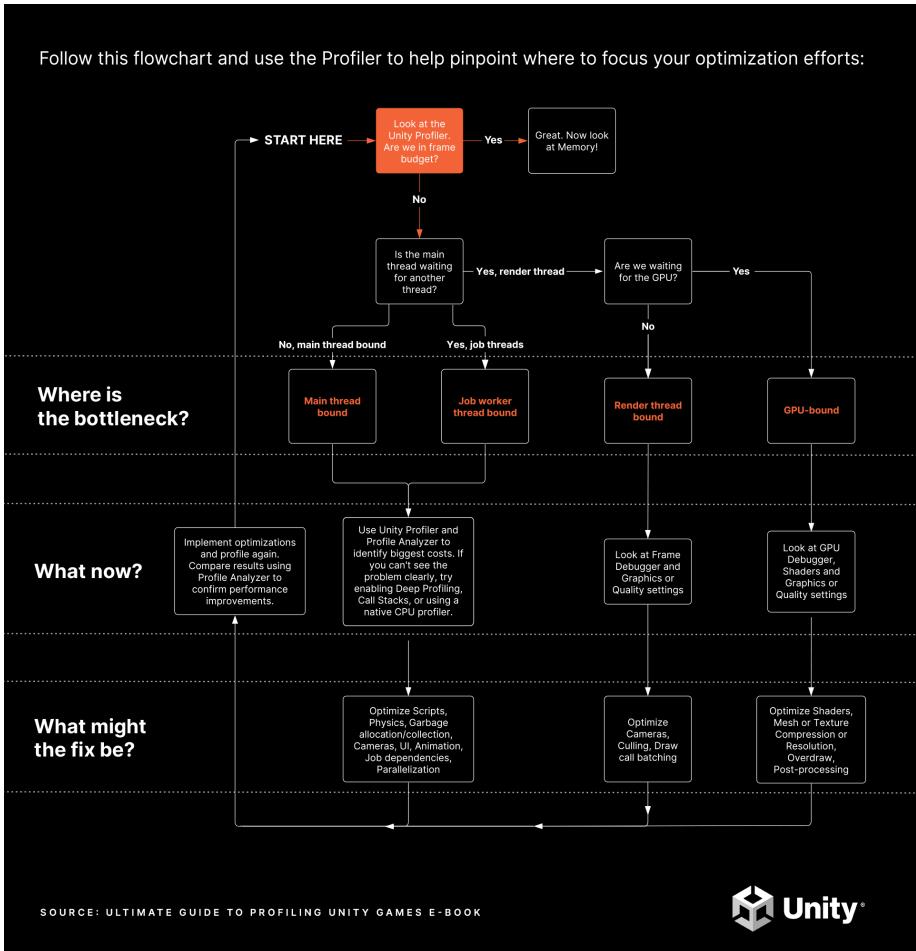
## Focus on optimizing the right areas

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. Profiling tools ultimately help you understand what's going on under the hood of your Unity project, but don't wait for significant performance problems to start showing before digging into your detective toolbox.

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.

To learn more about how to plan your profiling workflows see the [Ultimate guide to profiling Unity games](#).





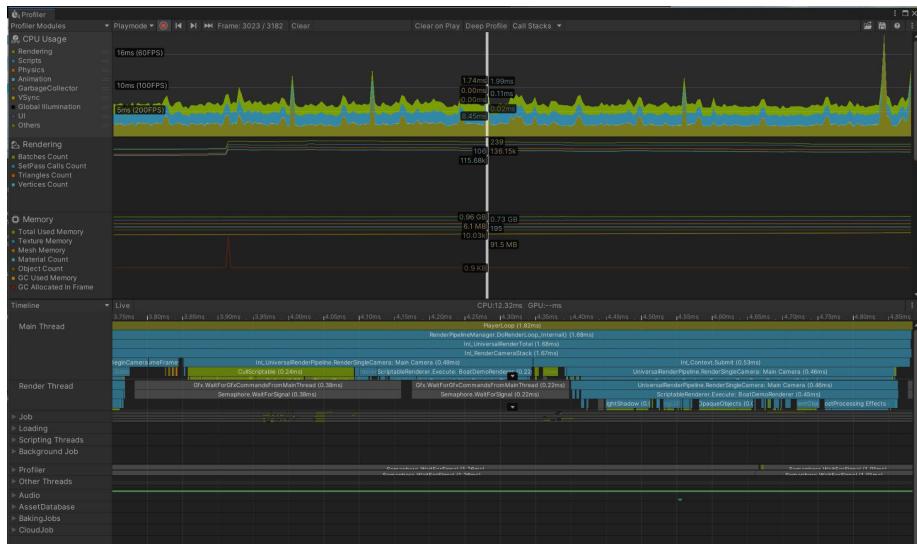
A chart from the profiling e-book featuring a workflow you can follow to profile your Unity projects efficiently.

## Understand how the Unity Profiler works

The built-in [Unity Profiler](#) can help you detect the causes of any lags or freezes at runtime and better understand what's happening at a specific frame or point in time.

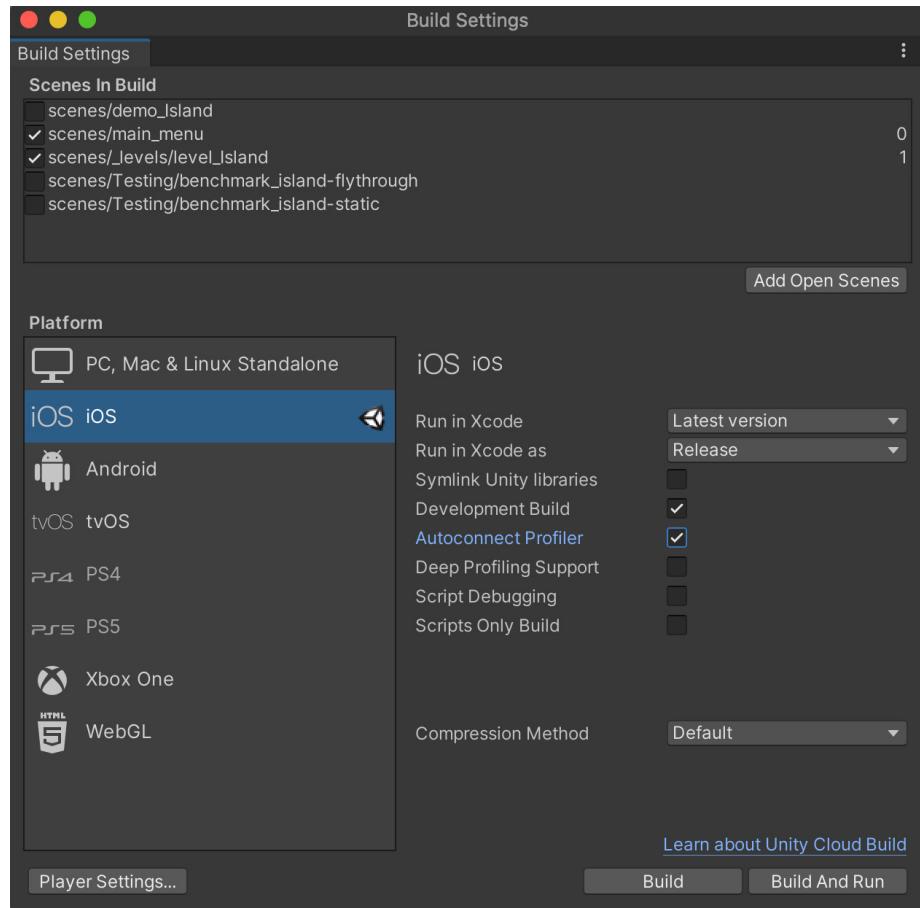
The Profiler is instrumentation-based; it profiles timings of game and engine code that are automatically marked up (such as MonoBehaviour's Start or Update methods, or specific API calls), or explicitly wrapped with the help of [ProfilerMarker](#) API.

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 (e.g., physics-heavy or music-based gameplay).



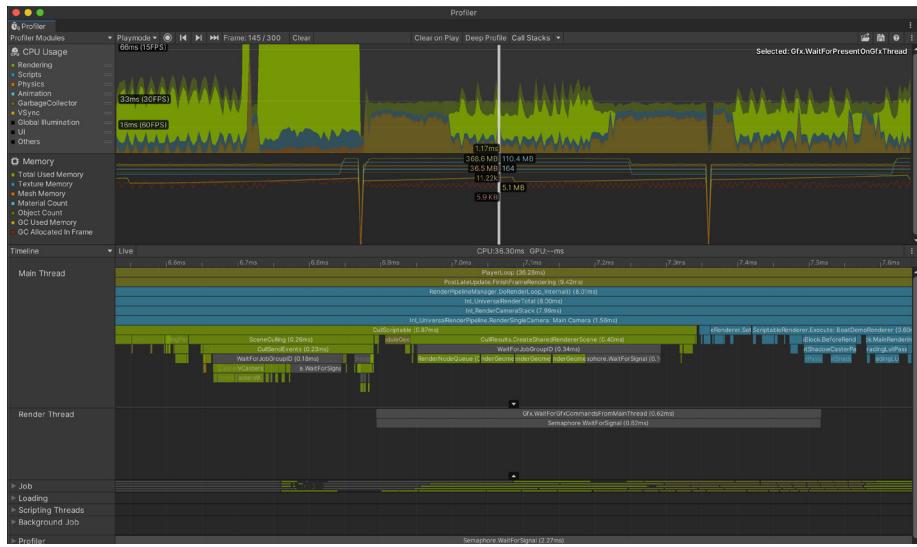
Use the Unity Profiler to test performance and resource allocation for your application.

To capture profiling data from an actual mobile device within your chosen platform, check the **Development Build** and **Autoconnect Profiler** boxes before you click **Build and Run**. Alternatively, if you want the app to start separately from your profiling, you can uncheck the **Autoconnect Profiler** box, and then connect manually once the app is running.



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 as far as 2000 if you need longer captures. While this means that the Unity Editor has to do more CPU work and take up more memory, it can be useful depending on your specific scenario.

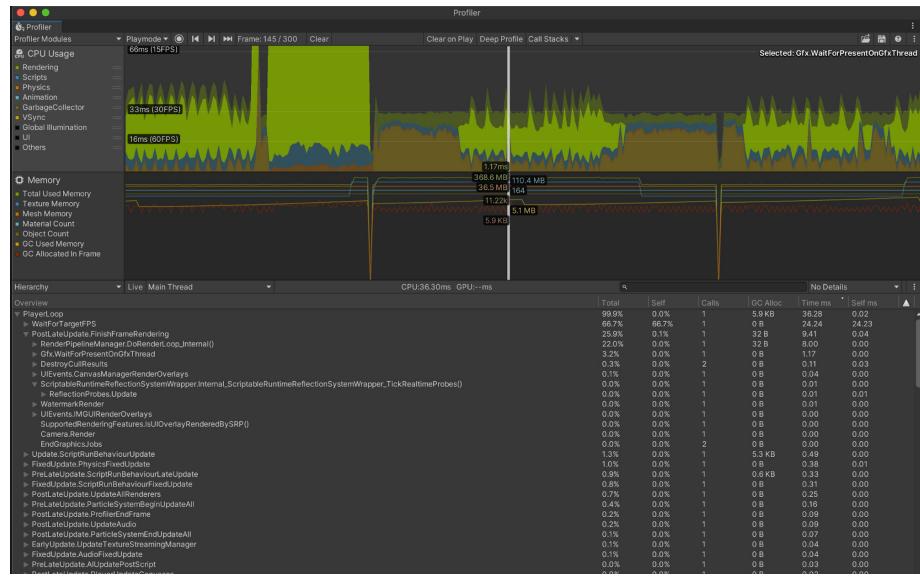


Use the Timeline view to determine if you are CPU-bound or GPU-bound.

When using the [Deep Profiling](#) setting, Unity can profile the beginning and end of every function call in your script code, telling you exactly which part of your application is being executed and potentially causing a delay. However, deep profiling adds overhead to every method call and may skew the performance analysis.

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.



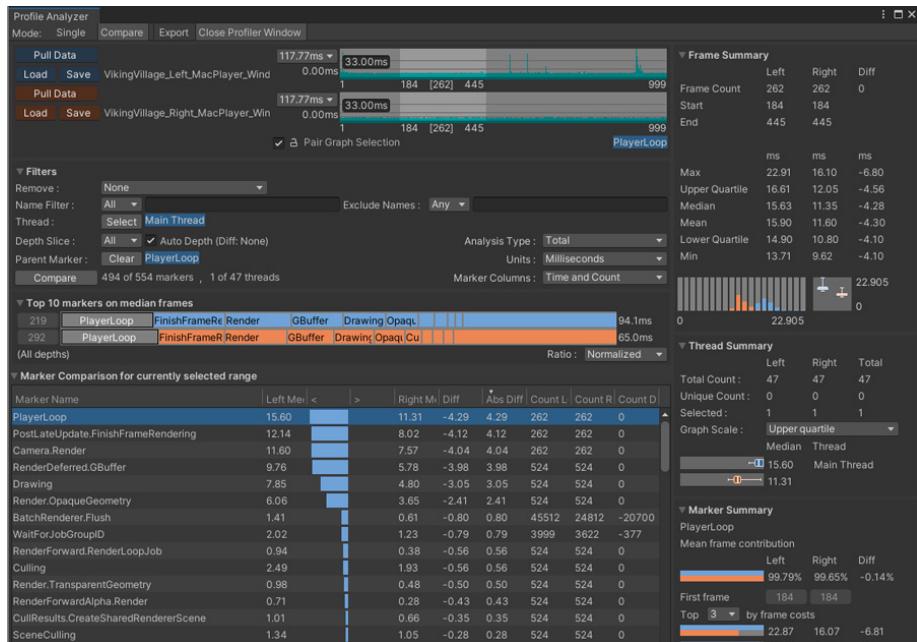
The Hierarchy view allows you to sort ProfileMarkers by time cost.

You can find a complete overview of the Unity Profiler [here](#). If you're new to profiling, you can also watch this [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. Then, rinse and repeat.

## Use the Profile Analyzer

The Profile Analyzer lets you aggregate multiple frames of Profiler data and then locate frames of interest. Do you want to see what happens to the Profiler after you make a change to your project? The **Compare** view allows you to load and differentiate two data sets, so you can test changes and improve their outcome. 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 frames per second (fps). For an application to run at 30 fps, its frame budget can't exceed 33.33 ms per frame ( $1000 \text{ ms} / 30 \text{ fps}$ ). Likewise, a target of 60 fps leaves 16.66 ms per frame.

## **Account for device temperature**

For mobile, however, we don't recommend using this maximum time consistently as the device can overheat and the OS can thermal throttle the CPU and GPU. We recommend that you use only about 65% of the available time to allow for cooldown between frames. A typical frame budget will be approximately 22 ms per frame at 30 fps and 11 ms per frame at 60 fps.

Devices can exceed this budget for short periods of time (e.g., for cutscenes or loading sequences) but not for a prolonged duration.

Most mobile devices do not have active cooling like their desktop counterparts. Physical heat levels can directly impact performance.

If the device is running hot, the Profiler might perceive and report poor performance, even if it is not cause for long-term concern. To combat profiling overheating, profile in short bursts. This cools the device and simulates real-world conditions. Our general recommendation is to keep the device cool for 10-15 minutes before profiling again.

## **Determine if you are GPU-bound or CPU-bound**

The central processing unit (CPU) is responsible for determining what must be drawn, and the graphics processing unit (GPU) is responsible for drawing it. When a rendering performance problem is due to the CPU taking too long to render a frame, the game becomes CPU bound. When a rendering performance problem is due to the GPU taking too long to render a frame, it becomes GPU 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, the render thread is ready, but you may be waiting for a bottleneck on the main thread.
- If you frequently encounter **Gfx.WaitForPresent**, that means the main thread was ready but was waiting for the GPU to present the frame.

## **Test on a min-spec device**

There is a wide range of iOS and Android devices out there. We want to reiterate the importance of testing your project on the minimum and maximum device specifications that you want your application to support, whenever possible.

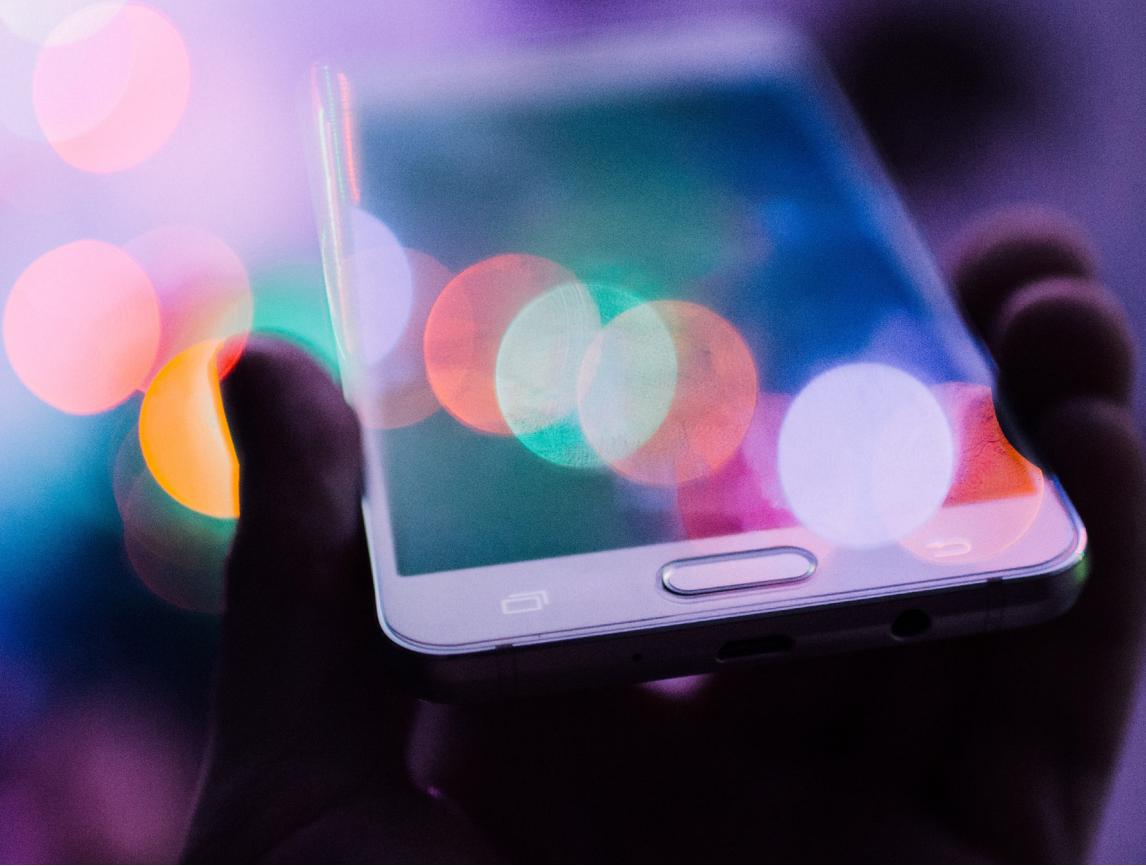
# MEMORY

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

The garbage collector periodically identifies and deallocates unused managed heap memory. The Asset garbage collection runs on demand or when you load a new scene and deallocates native objects and resources. 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 heap memory, and how you minimize the effect of garbage collection.

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

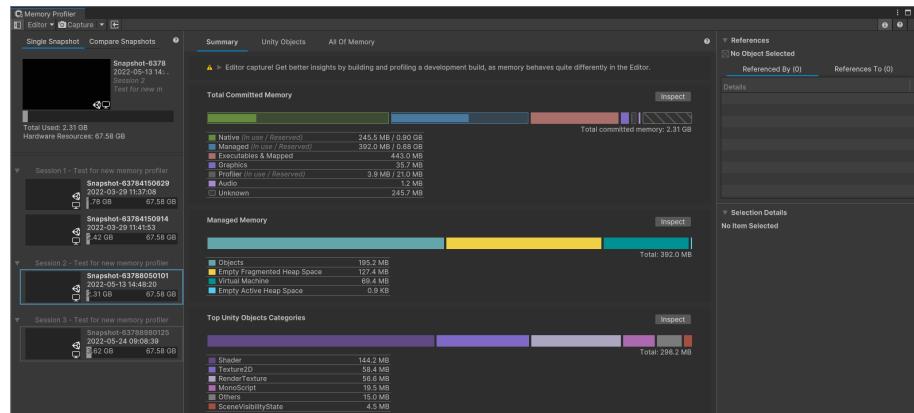


## Use the Memory Profiler

The [Memory Profiler package](#) takes a snapshot of your managed heap memory to help you identify problems like fragmentation and memory leaks.

Use the **Unity Objects** tab to identify areas where you can eliminate duplicate memory entries or find which objects use the most memory. The **All of Memory** tab displays a breakdown of all the memory in the snapshot that Unity tracks.

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



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

## 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. Reduce unnecessary string creation or manipulation. Avoid parsing string-based data files such as JSON and XML; store data in ScriptableObjects or formats such as MessagePack or Protobuf instead. Use the [StringBuilder](#) class if you need to build strings at runtime.
- **Unity function calls:** Be aware that some functions create heap allocations. 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](#) (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 (e.g., `int i = 123; object o = i`) implicitly converts the value type to a type object. Instead, try to provide concrete overrides with the value type you want to pass in. Generics can also be used for these overrides.
- **Couroutines:** Though `yield` does not produce garbage, creating a new [WaitForSeconds](#) object does. Cache and reuse the [WaitForSeconds](#) object instead of 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.

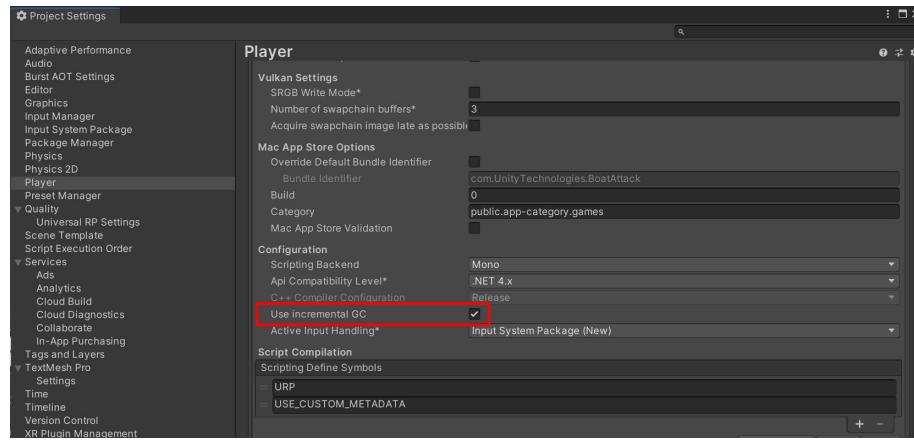
For more information, see the manual page on [Garbage collection best practices](#).

## Time garbage collection if 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.<sup>2</sup>

<sup>2</sup>Note 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 reduce GC spikes.

### Use the incremental garbage collector to split the GC workload

Instead of using a single, long interruption of your program's execution, incremental garbage collection uses multiple, much-shorter interruptions, distributing the workload over many frames. If garbage collection is impacting performance, try enabling this option to see if it can significantly reduce the problem of GC spikes. Use the Profile Analyzer to verify the benefit to your application.

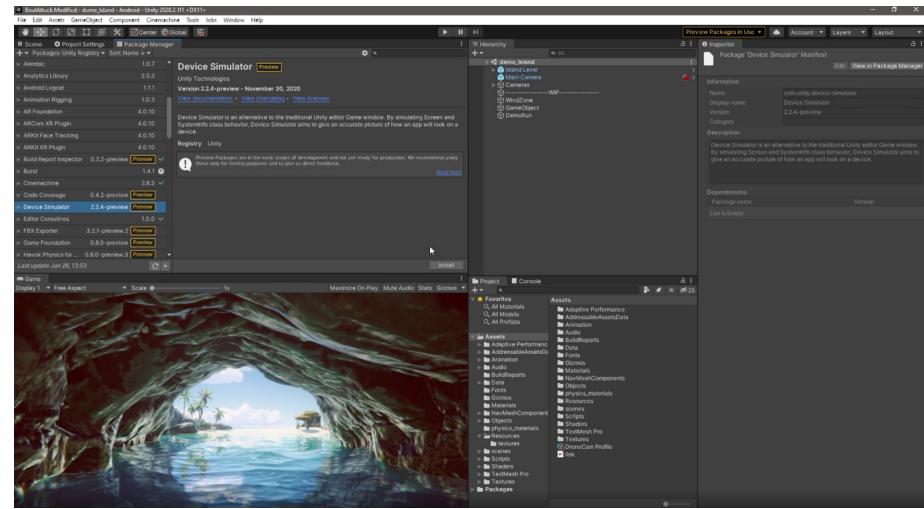
# ADAPTIVE PERFORMANCE

With Unity and Samsung's [Adaptive Performance](#), you can monitor the device's thermal and power state to ensure that you're ready to react appropriately. When users play for an extended period of time, you can reduce your level of detail (or LOD) bias dynamically to help your game continue to run smoothly. Adaptive Performance allows developers to increase performance in a controlled way while maintaining graphics fidelity.

While you can use Adaptive Performance APIs to fine-tune your application, this package also offers automatic modes. In these modes, Adaptive Performance determines the game settings along several key metrics:

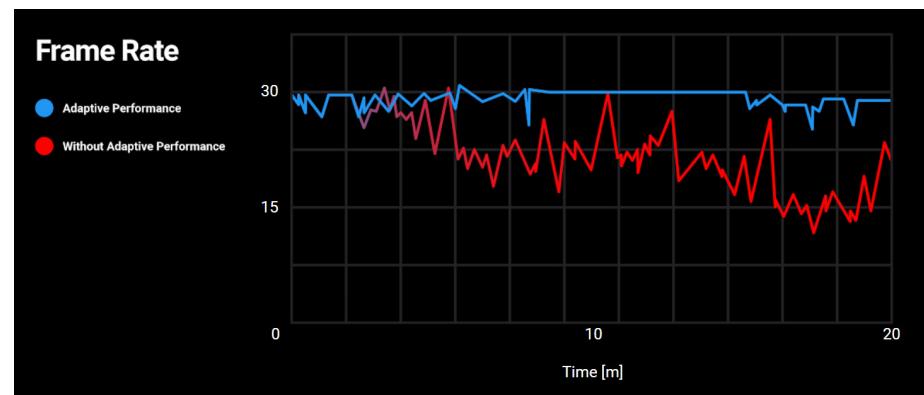
- Desired frame rate based on previous frames
- Device temperature level
- Device proximity to thermal event
- Device bound by CPU or GPU

These four metrics dictate the state of the device, and Adaptive Performance tweaks the adjusted settings to reduce the bottleneck. This is done by providing an integer value, known as an Indexer, to describe the state of the device.



Note that Adaptive Performance only works for Samsung devices.

To learn more about Adaptive Performance, you can view the [samples](#) we've provided in the Package Manager by selecting **Package Manager > Adaptive Performance > Samples**. Each sample interacts with a specific scaler, so you can see how the different scalers impact your game. We also recommend reviewing the [end-user documentation](#) to learn more about Adaptive Performance configurations and how you can interact directly with the API.



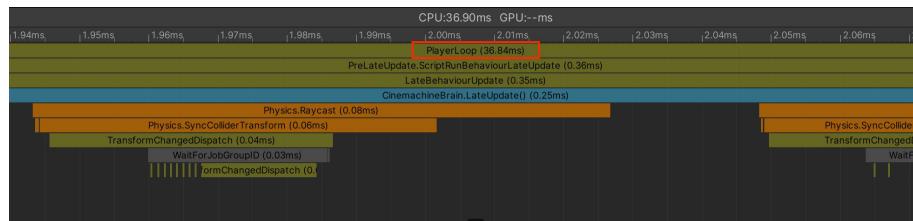
Note that Adaptive Performance only works for Samsung devices.



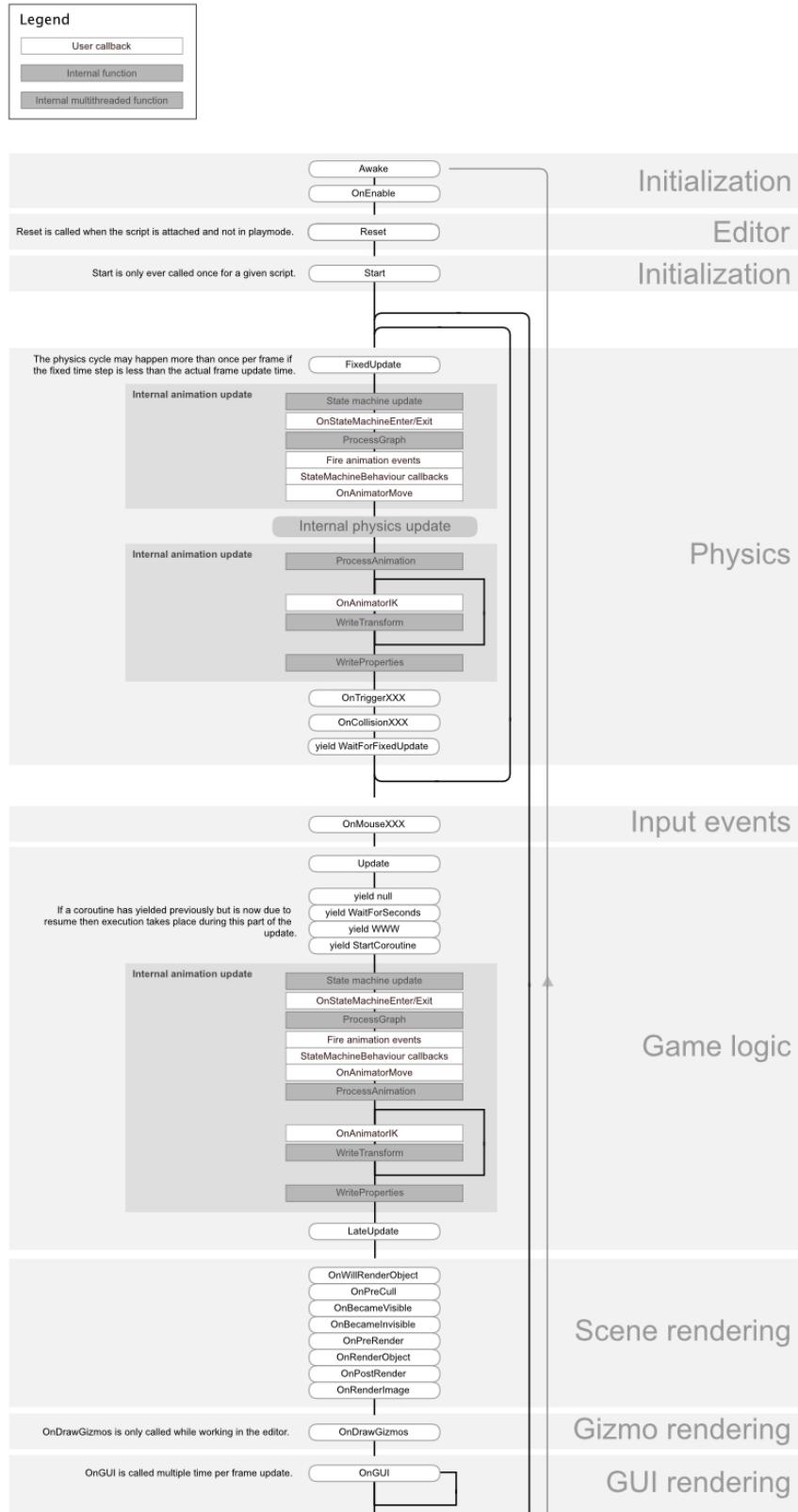
# PROGRAMMING AND CODE ARCHITECTURE

The Unity [PlayerLoop](#) contains functions for interacting with the core of the game engine. This tree-like 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 all of 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.



Get to know the PlayerLoop and the [lifecycle of a script](#).

You can optimize your scripts with these tips and tricks.

### 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. You can utilize the Low-Level API to add custom logic to the player's update loop.

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

### 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, but extract 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 of applying 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();
    }
}
```

Better yet, if **ExampleExpensiveFunction** performs some operation on a set of data, consider using time slicing to operate on a different subset of that data every frame. By doing  $1/n$  of the work every frame rather than all of the work every  $n$  frames, you end up with performance that is more stable and predictable overall, rather than seeing periodic CPU spikes.

## Avoid heavy logic in Start/Awake

When your first scene loads, these functions get called for each object:

- **Awake**
- **OnEnable**
- **Start**

Avoid expensive logic in these functions until your application renders its first frame. Otherwise, you may encounter longer loading times than necessary.

Refer to the [order of execution for event functions](#) for details about the first scene load.

## Avoid empty Unity events

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

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

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

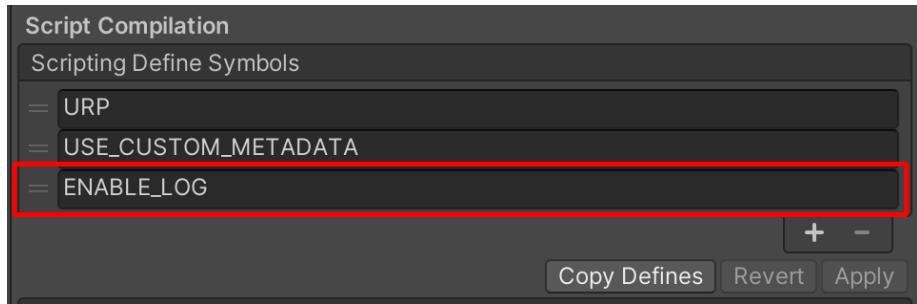
Here, you can freely use the **Update** in the Editor for testing without unnecessary overhead slipping into your build.

## 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**, all of your **Log** statements disappear in one fell swoop.

The same thing applies for other use cases of the Debug Class, such as Debug.DrawLine and Debug.DrawRay. These are also only intended for use during development and can significantly impact performance.

### 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. Get these hashes during initialization and cache them in variables for when they're needed to pass to a Get or Set method.

### Choose the right data structure

Your choice of data structure can have a cumulative effect on efficiency or inefficiency as you iterate many thousands of times per frame. Does it make more sense to use a List, Array, or Dictionary for your collection? Follow the [MSDN guide to data structures](#) in C# as a general guide to choosing the correct structure.

### Avoid adding components at runtime

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

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

## Cache GameObjects and components

**GameObject.Find**, **GameObject.GetComponent**, and **Camera.main** (in versions prior to 2020.2) can be expensive, so avoid calling them in **Update** methods. Instead, call them in **Start** and cache the results.

For example, this demonstrates inefficient use of a repeated **GetComponent** call:

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

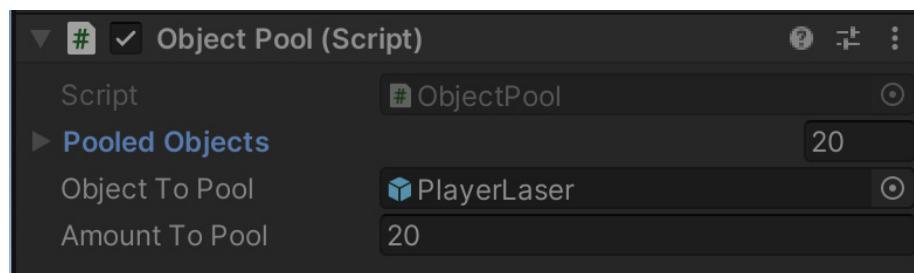
It's more efficient to invoke **GetComponent** only once, as the result of the function is cached. The cached result can be reused in **Update** without any further calls to **GetComponent**.

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

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

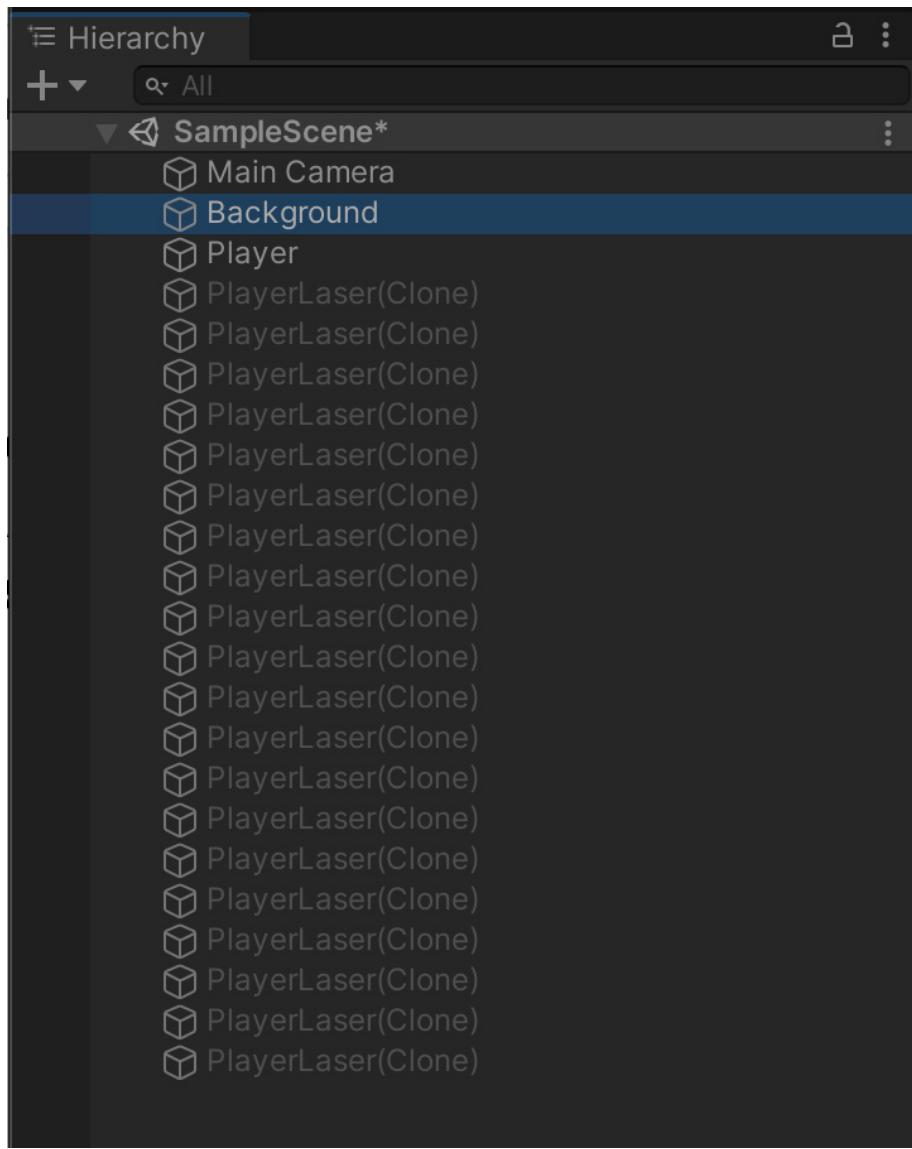
## Use object pools

**Instantiate** and **Destroy** can generate garbage and garbage collection (GC) spikes and is generally a slow process.



In this example, the ObjectPool creates 20 PlayerLaser instances for reuse.

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



The pool of PlayerLaser objects is inactive and ready to shoot.

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](#). You can also see [these code examples](#), from Unity, using Object pooling to inform your own game development.

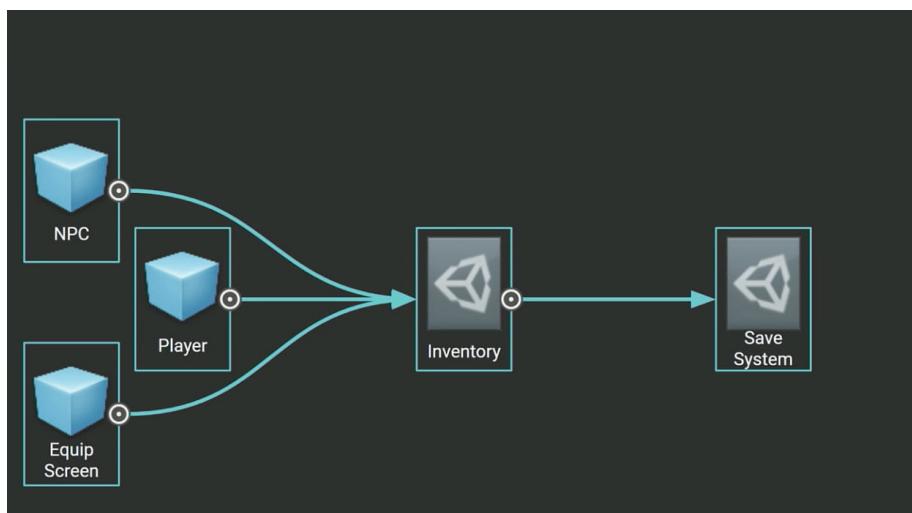
## Use ScriptableObjects

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.

MonoBehaviours carry extra overhead since they require a GameObject – and by default a Transform – to act as a host. That means you need to create a lot of unused data before storing a single value. The ScriptableObject slims down this memory footprint by dropping the GameObject and Transform. It also stores the data at the project level, which is helpful if you need to access the same data from multiple scenes.

A common use case is having many GameObjects that rely on the same duplicate data that does not need to change at runtime. Rather than having this duplicate local data on each GameObject, you can funnel it into a ScriptableObject. Then, each of the objects stores a reference to the shared data asset, rather than copying the data itself. This can provide significant performance improvements in projects with thousands of objects.

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

In software design, this is an optimization known as the [flyweight pattern](#). Restructuring your code in this way using `ScriptableObjects` avoids copying a lot of values and reduces your memory footprint.

Watch this [Introduction to ScriptableObjects](#) devlog to see how `ScriptableObjects` can benefit your project. You can also find relevant documentation [here](#) or our technical guide [Create modular game architecture in Unity with ScriptableObjects](#).

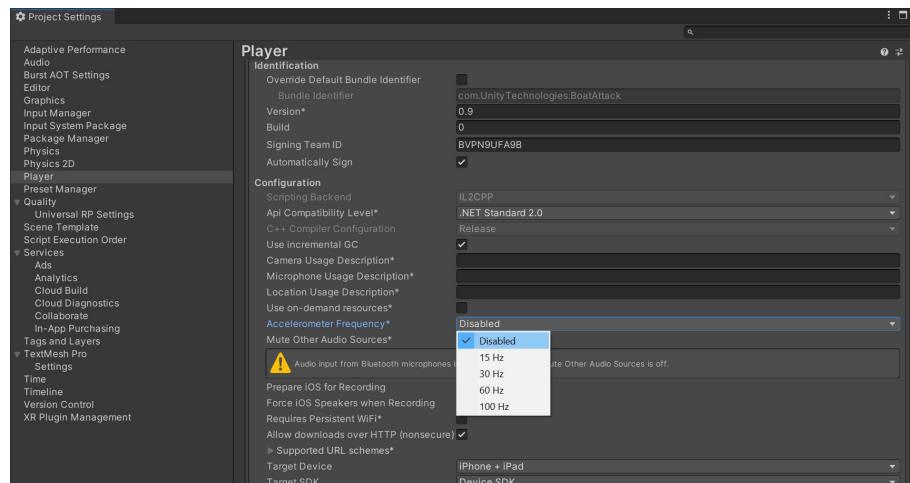
To learn more about using design patterns in Unity, see the e-book [Level up your code with game programming patterns](#) to learn more about using design patterns.

# PROJECT CONFIGURATION

There are a few Project Settings that can impact your mobile performance.

## Reduce or disable Accelerometer Frequency

Unity pools your mobile's accelerometer several times a second.  
Disable this if it's not being used in your application, or reduce its frequency  
for better performance.



Ensure your Accelerometer Frequency is disabled if you are not making use of it in your mobile game.

## Disable unnecessary Player or Quality settings

In the **Player** settings, disable **Auto Graphics API** for unsupported platforms to prevent generating excessive shader variants. Disable **Target Architectures** for older CPUs if your application is not supporting them.

In the **Quality** settings, disable unneeded Quality levels.

## Disable unnecessary physics

If your game is not using physics, uncheck **Auto Simulation** and **Auto Sync Transforms**. These will just slow down your application with no discernible benefit.

## Choose the right frame rate

Mobile projects must balance frame rates against battery life and thermal throttling. Instead of pushing the limits of your device at 60 fps, consider running at 30 fps as a compromise. Unity defaults to 30 fps for mobile.

You can also adjust the frame rate dynamically during runtime with **Application.targetFrameRate**. For example, you could even drop below 30 fps for slow or relatively static scenes and reserve higher fps settings for gameplay.

## 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](#)” on the Unity Blog and this [Unite talk](#) for best practices with Transforms.

## Transform once, not twice

Also, 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 details about Object.Instantiate, please see the [Scripting API](#).

### **Assume Vsync is enabled**

Mobile platforms won't render half-frames. Even if you disable Vsync in the Editor (**Project Settings > Quality**), Vsync is enabled at the hardware level. If the GPU cannot refresh fast enough, the current frame will be held, effectively reducing your fps.

# ASSETS

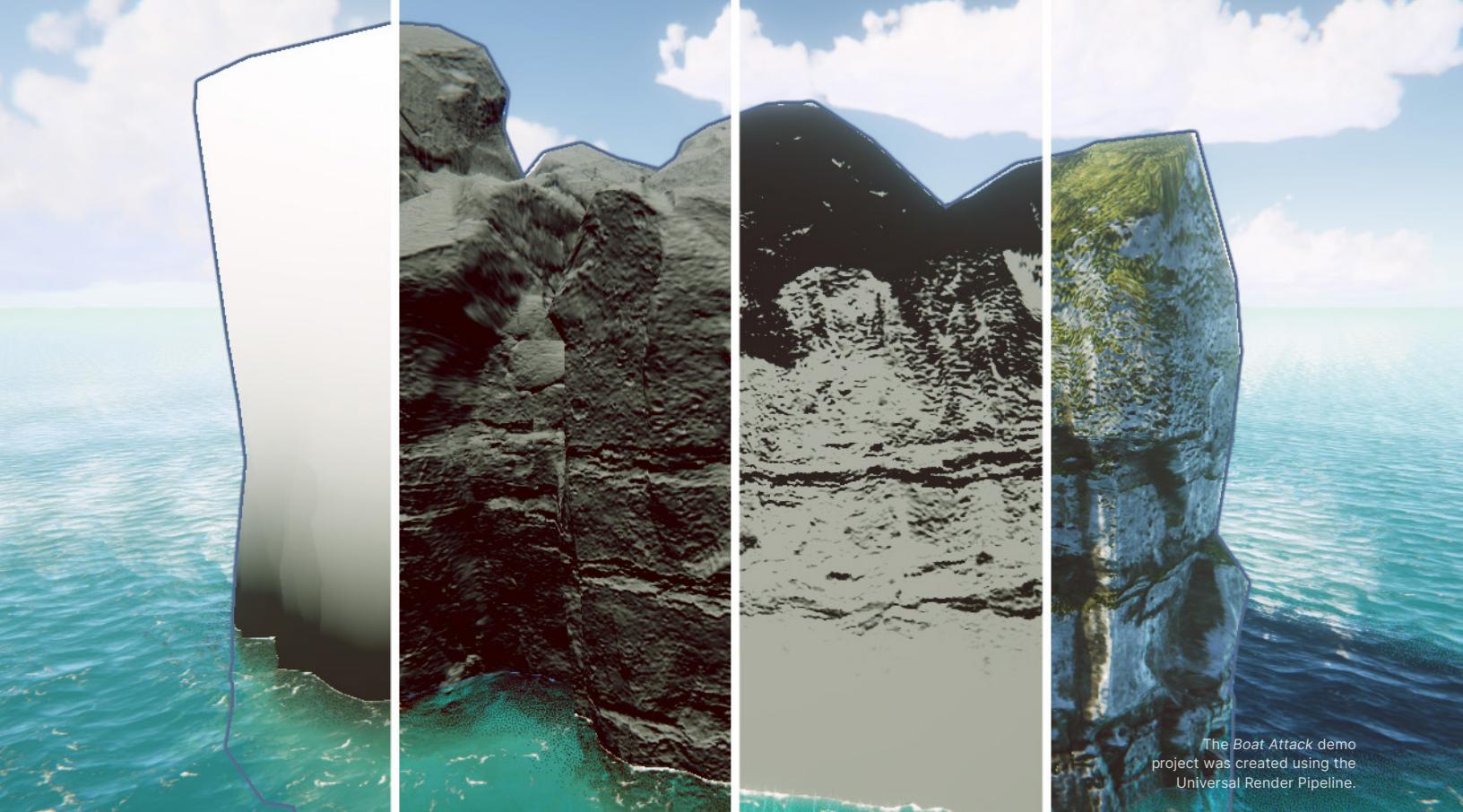
The asset pipeline can dramatically impact your application's performance. An experienced technical artist can help your team define and enforce asset formats, specifications, and import settings.

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

See [this guide to best practices for art assets](#) for more detail or check out this course about [3D Art Optimization for Mobile Applications](#) on Unity Learn for more details.

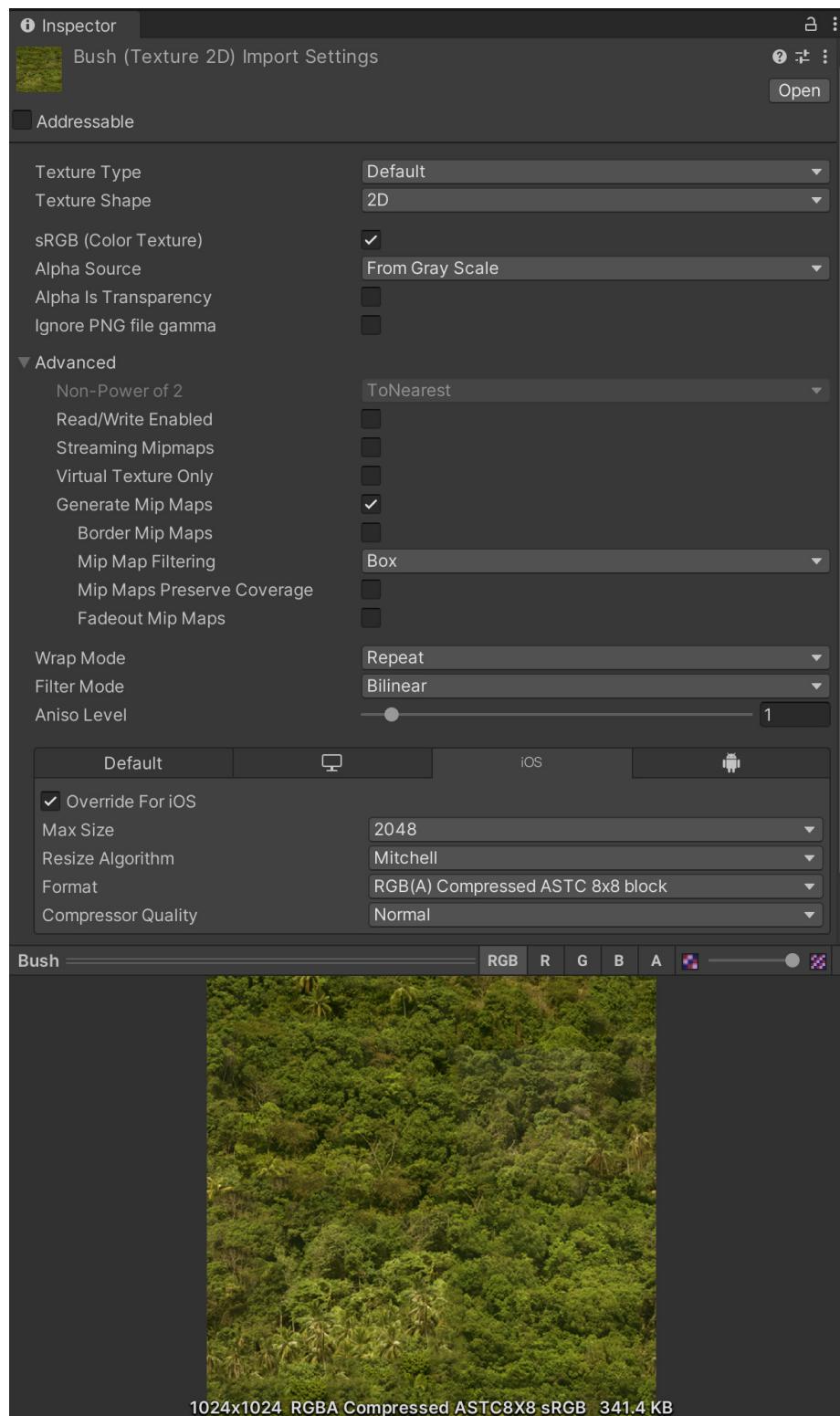
## **Import textures correctly**

Most of your memory will likely go to textures, so the import settings here are critical. In general, follow these guidelines:



The Boat Attack demo project was created using the Universal Render Pipeline.

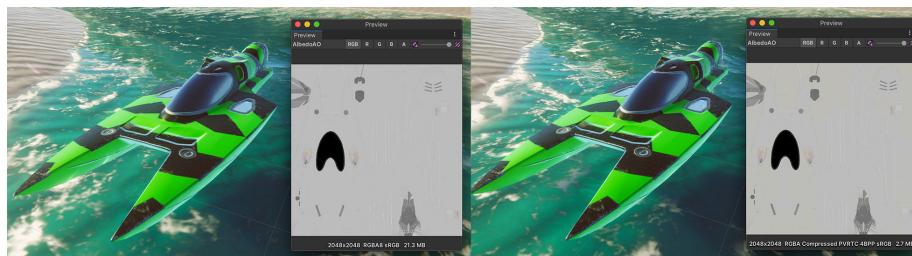
- **Lower the Max Size:** Use the minimum settings that produce visually acceptable results. This is non-destructive and can quickly reduce your texture memory.
- **Use powers of two (POT):** Unity requires POT texture dimensions for mobile texture compression formats (PVRCT or ETC).
- **Atlas your textures:** Placing multiple textures into a single texture can reduce draw calls and speed up rendering. Use the [Unity Sprite Atlas](#) or the third-party [Texture Packer](#) to atlas your textures.
- **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. If you are generating textures at runtime, enforce this via [`Texture2D.Apply`](#), passing in [`makeNoLongerReadable`](#) set to **true**.
- **Disable unnecessary Mip Maps:** Mip Maps are not needed for textures that remain at a consistent size on-screen, such as 2D sprites and UI graphics (leave Mip Maps enabled for 3D models that vary their distance from the camera).



Proper texture import settings will help optimize your build size.

## Compress textures

Consider these two examples using the same model and texture. The settings on the left consume almost eight times the memory as those on the right, without much benefit in visual quality.



Uncompressed textures require more memory.

Use Adaptive Scalable Texture Compression (ASTC) for both iOS and Android. The vast majority of games in development target min-spec devices that support ASTC compression.

The only exceptions are:

- iOS games targeting A7 devices or lower (e.g., iPhone 5, 5S, etc.)
  - use PVRTC
- Android games targeting devices prior to 2016 – use ETC2 (Ericsson Texture Compression)

If the quality of compressed formats such as PVRTC and ETC isn't sufficiently high, and if ASTC is not fully supported on your target platform, try using 16-bit textures instead of 32-bit textures.

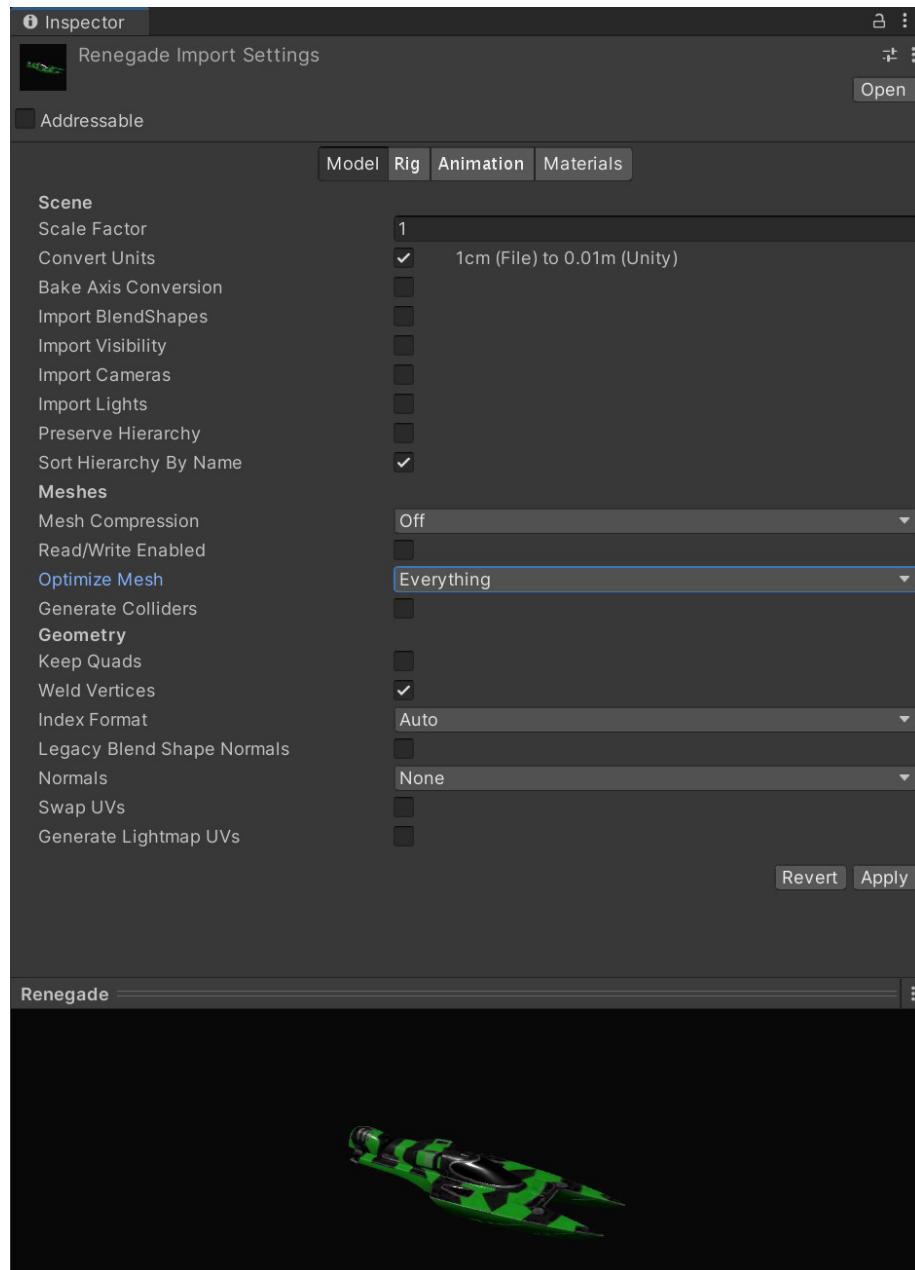
See the manual for more information on [recommended texture compression format by platform](#).

## Adjust mesh import settings

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

- **Compress the mesh:** Aggressive 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, keeping 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 possible:** If you are certain the mesh's material will not need normals or tangents, uncheck these options for extra savings.



Check your mesh import settings.

## **Check your polygon counts**

Higher-resolution models mean more memory usage and potentially longer GPU times. Does your background geometry need half a million polygons? Consider cutting down models in your DCC package of choice. Delete unseen polygons from the camera's point of view. Use textures and normal maps for fine detail instead of high-density meshes.

## **Automate your import settings using the AssetPostprocessor**

The [AssetPostprocessor](#) allows you to hook into the import pipeline and run scripts prior to or when importing assets. This prompts you to customize settings before and/or after importing models, textures, audio, and so on in a way similar to presets but through code. Learn more about the process in our GDC 2023 talk, "[Technical tips for every stage of game creation](#)."

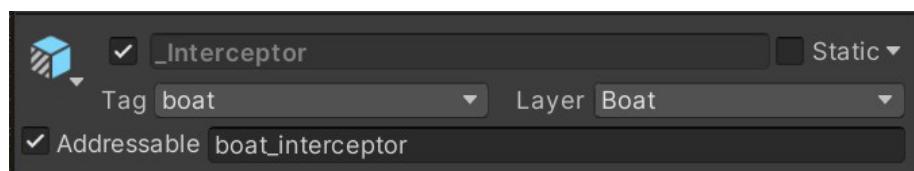
## **Unity DataTools**

[Unity DataTools](#) is a collection of open source tools provided by Unity that aims to enhance data management and serialization capabilities in Unity projects. It includes features for analyzing and optimizing project data, such as identifying unused assets, detecting asset dependencies, and reducing build size.

## **Use the Addressable Asset System**

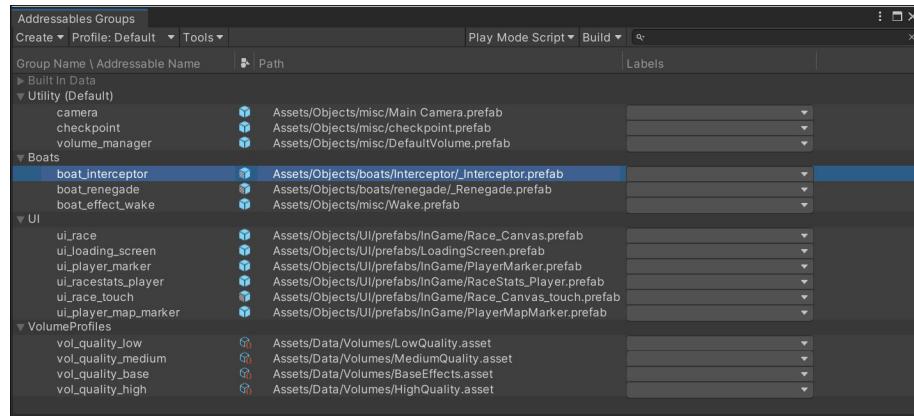
The [Addressable Asset System](#) provides a simplified way to manage your content, loading AssetBundles by "address" or alias. This unified system loads asynchronously from either a local path or a remote content delivery network (CDN).

If you split your non-code assets (models, textures, Prefabs, audio, and even entire scenes) into an [AssetBundle](#), you can separate them as downloadable



content (DLC).

Then, use Addressables to create a smaller initial build for your mobile application. [Cloud Content Delivery](#) lets you host and deliver your game content to players as they progress through the game.



Load assets by "address" using the Addressable Asset System.

Click [here](#) to see how the Addressable Asset System can take the pain out of asset management.

# GRAPHICS AND GPU OPTIMIZATION

With each frame, Unity determines the objects that must be rendered and then creates draw calls. A draw call is a call to the graphics API to draw objects (e.g., a triangle), whereas a batch is a group of draw calls to be executed together.

As your projects become more complex, you'll need a pipeline that optimizes the workload on your GPU. [The Universal Render Pipeline](#) (URP) supports three options for rendering: Forward, Forward+, and Deferred.



*Boat Attack* demo project created using the Universal Render Pipeline.

Forward rendering evaluates all lighting in a single pass and is generally recommended as default for mobile games. Forward+, introduced with Unity 2022 LTS, improves upon standard Forward rendering by culling lights spatially rather than per object. This significantly increases the overall number of lights that can be utilized in rendering a frame. Deferred mode is a good choice for specific cases, such as for games with lots of dynamic light sources. The same physically based lighting and materials from consoles and PCs can also scale to your phone or tablet.

The following table compares the three rendering options in URP.

Feature	Forward	Forward+	Deferred
Maximum number of real-time lights per object	9	Unlimited; <a href="#">the per-Camera limit applies</a>	Unlimited
Per pixel normal encoding	No encoding (accurate normal values)	No encoding (accurate normal values)	Two options: <ul style="list-style-type: none"><li>— Quantization of normals in G-buffer (loss of accuracy, better performance)</li><li>— Octahedron encoding (accurate normals, might have significant performance impact on mobile GPUs)</li></ul> For more information, see <a href="#">Encoding of normals in G-buffer</a> .
MSAA	Yes	Yes	No
Vertex lighting	Yes	No	No
Camera stacking	Yes	Yes	Supported with a limitation: Unity renders only the base Camera using the Deferred path; Unity renders all overlay Cameras using the Forward Rendering path

Learn about moving projects based on the Built-in Render Pipeline to URP with the e-book [Introduction to the Universal Render Pipeline for advanced Unity creators](#).

## GPU optimization

To 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 onscreen; 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.

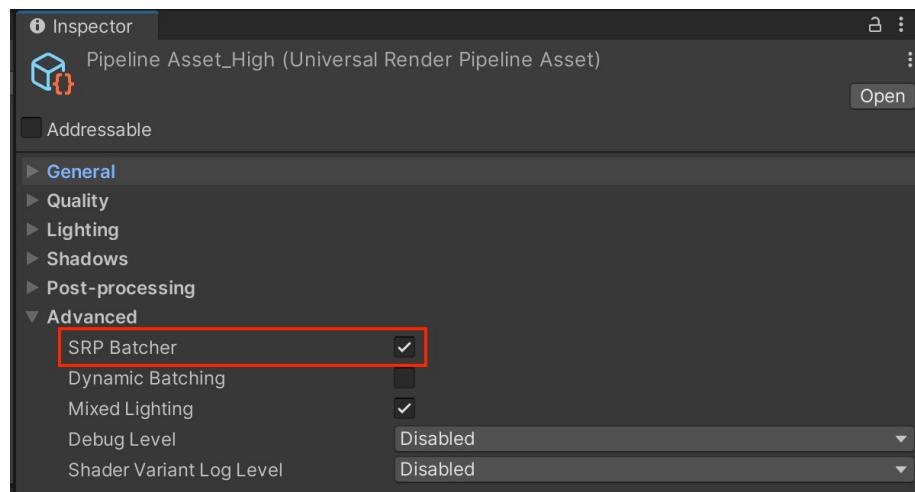
## Use draw call batching

To draw a GameObject, 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 draw calls 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, and so on 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.

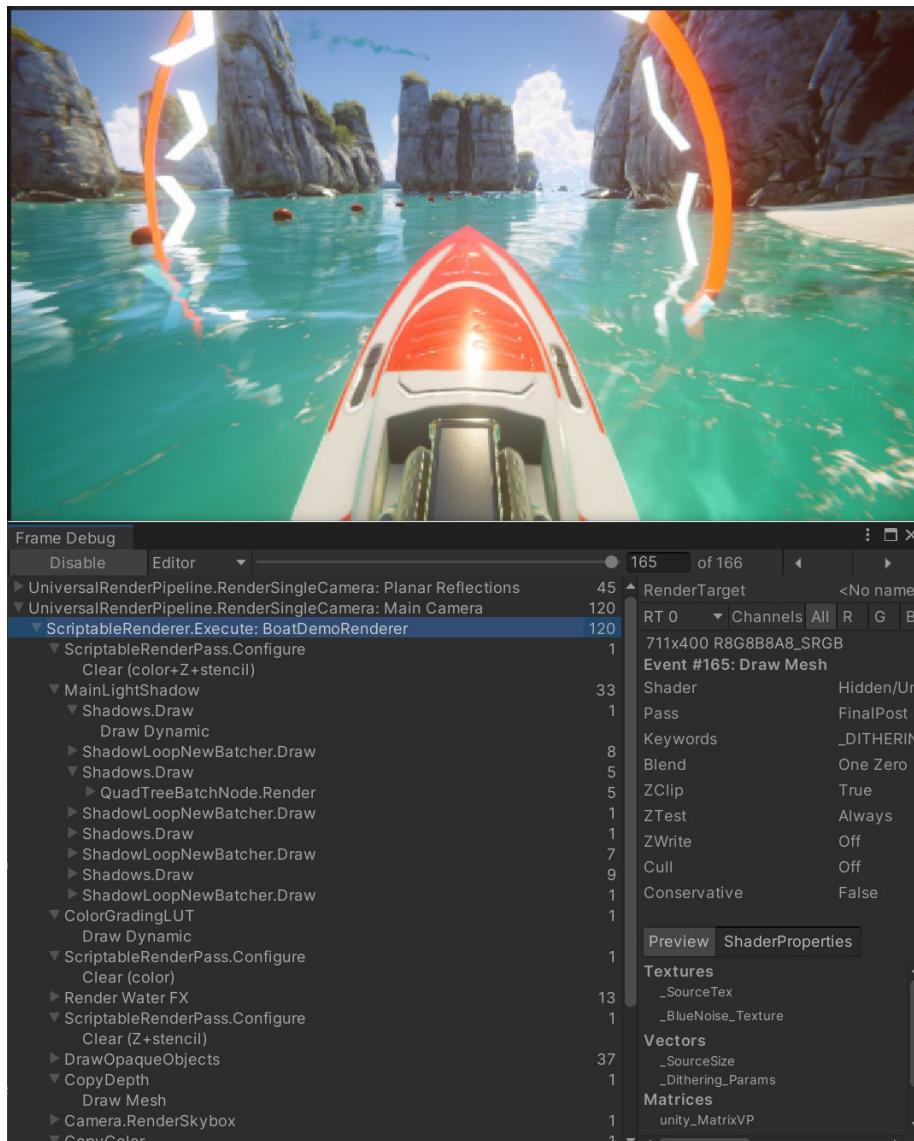
You can maximize batching with a few simple rules:

- 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.

## Use the Frame Debugger

The [Frame Debugger](#) shows how each frame is constructed from individual draw calls. This is an invaluable tool for troubleshooting your shader properties and can help you analyze how the game is rendered.



The Frame Debugger breaks each frame into its separate steps.

New to the Frame Debugger? Check out this introduction [here](#).

## Avoid too many dynamic lights

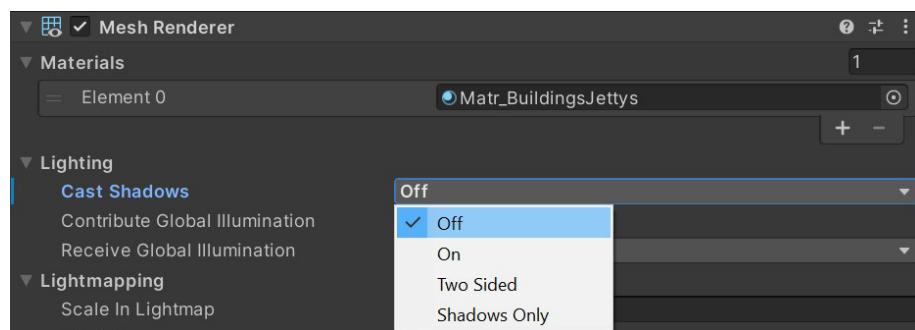
It is crucial to avoid adding too many dynamic lights to your mobile application when using forward rendering. Consider alternatives like custom shader effects and light probes for dynamic meshes, as well as baked lighting for static meshes.

See this [feature comparison table](#) for the specific limits of URP and Built-in Render Pipeline real-time lights.

## Disable shadows

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

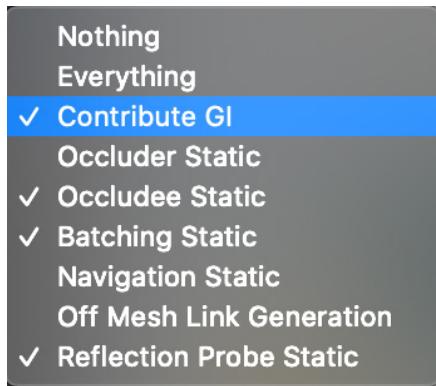
You can also create fake shadows using a blurred texture applied to a simple mesh or quad underneath your characters. Alternately, create blob shadows with custom shaders.



Disable shadow casting to reduce draw calls.

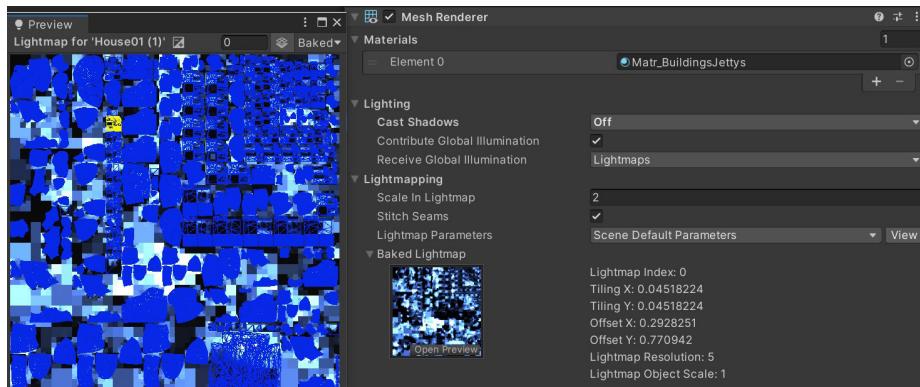
## Bake your lighting into Lightmaps

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.



Enable Contribute GI.

Baked shadows and lighting can then render without a performance hit at runtime. The Progressive CPU and GPU Lightmapper can accelerate the baking of Global Illumination.

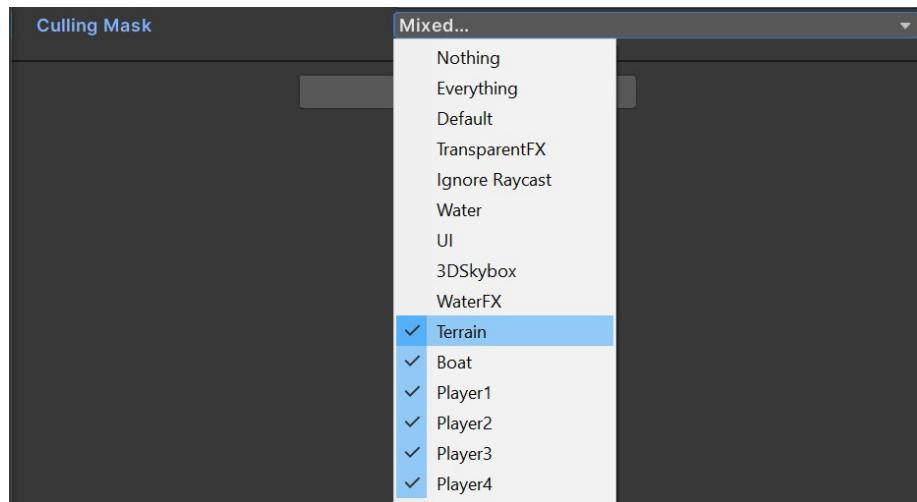


Adjust the Lightmapping Settings ([Windows > Rendering > Lighting Settings](#)) and Lightmap size to limit memory usage.

Follow the [manual guide](#) and [this article on light optimization](#) to get started with Lightmapping in Unity.

## Use Light Layers

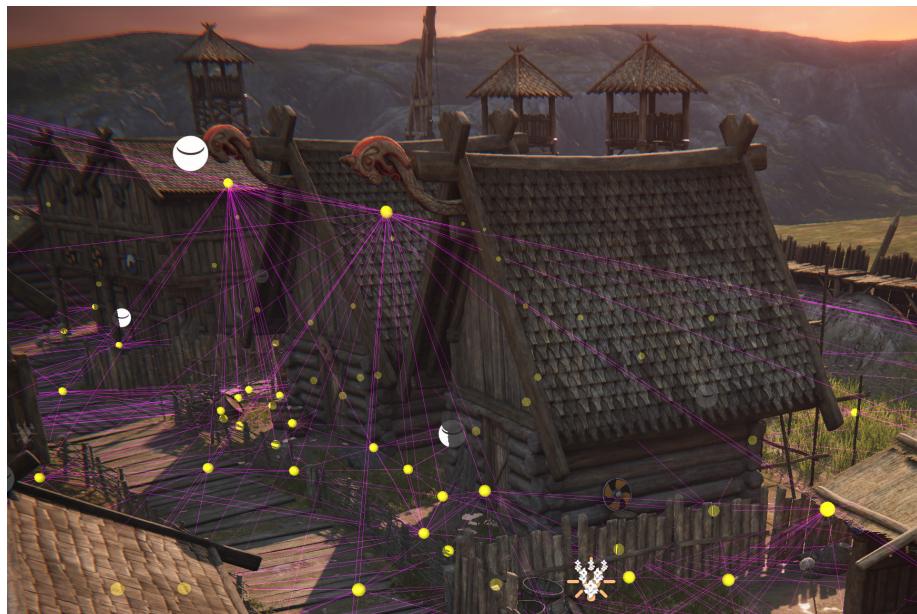
For complex scenes with multiple lights, separate your objects using layers, then confine each light's influence to a specific culling mask.



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

## Use Light Probes for moving 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 quickly compared to dynamic lights. This is especially useful for moving objects that normally cannot receive baked lightmapping.



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

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**.

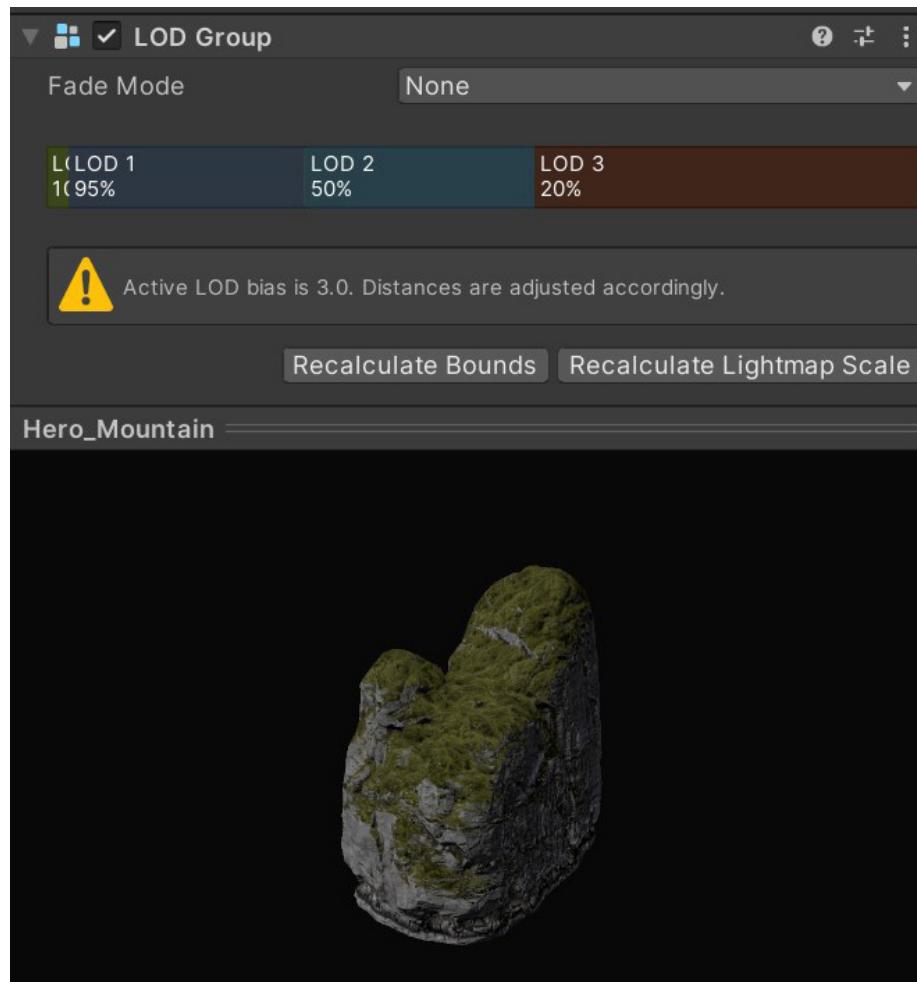
Continue using lightmapping for your prominent level geometry, but use probes for smaller details. Light Probe illumination does not require proper UVs, saving you the extra step of unwrapping your meshes. Probes also reduce disk space since they don't generate lightmap textures.

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

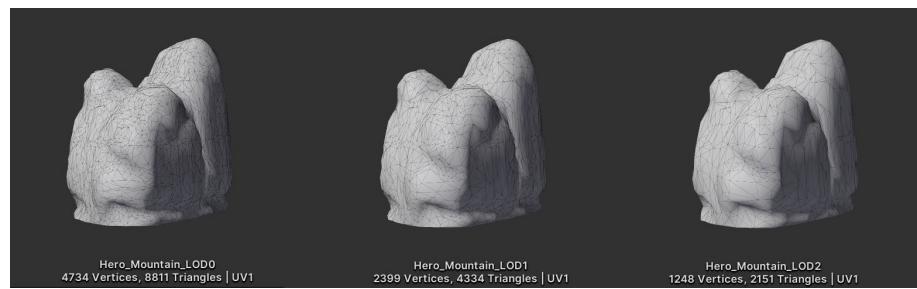
For more about lighting workflows in Unity, read "[Making believable visuals in Unity](#)."

## Use Level of Detail (LOD)

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



Example of a mesh using a LOD Group.



Source meshes, modeled at varying resolutions.

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

## **Use Occlusion Culling to remove hidden objects**

Objects hidden behind other objects can potentially still render and cost resources. Use Occlusion Culling to discard them.

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 via **Window > Rendering > Occlusion Culling**. Though not necessary for every scene, culling can improve performance in specific cases, so be sure to profile before and after enabling occlusion culling to check if it has improved performance.

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

## **Avoid mobile native resolution**

Phones and tablets have become increasingly advanced, with newer devices sporting very high resolutions.

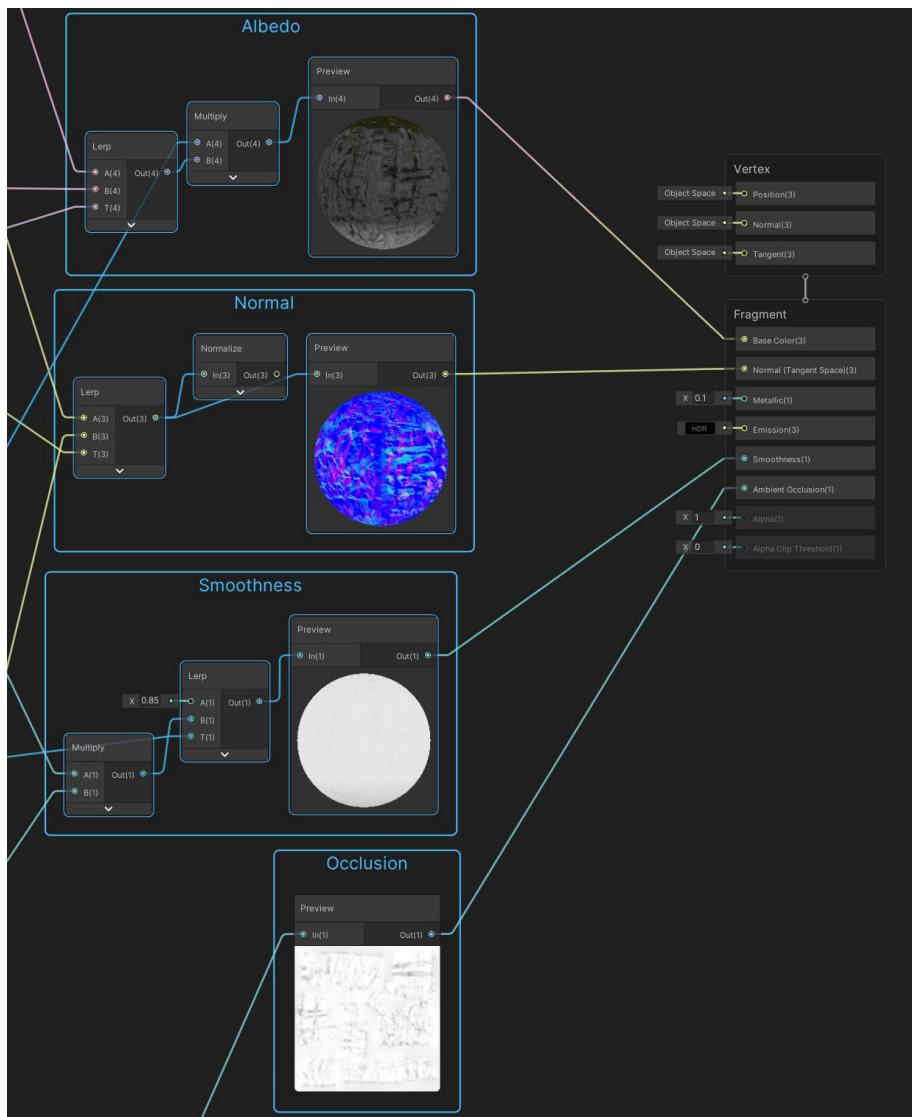
Use **Screen.SetResolution(width, height, false)** to lower the output resolution and regain some performance. Profile multiple resolutions to find the best balance between quality and speed.

## **Limit use of cameras**

Each enabled camera incurs some overhead, whether it's doing meaningful work or not. Only use camera components necessary for rendering. On lower-end mobile platforms, each camera can use up to 1 ms of CPU time.

## Keep shaders simple

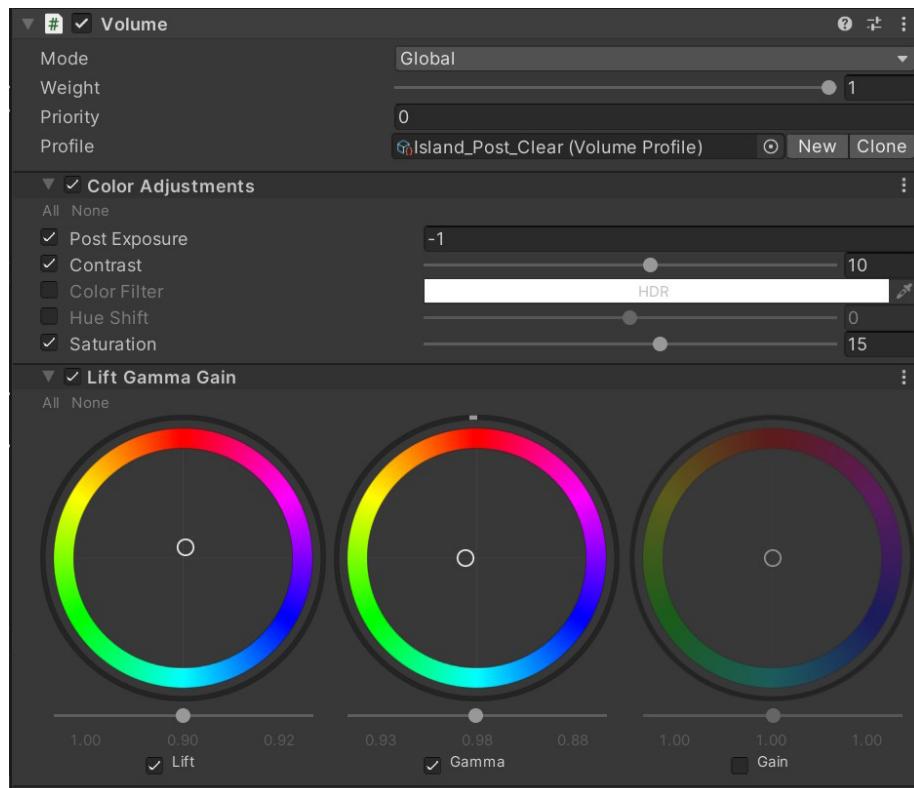
The Universal Render Pipeline includes several lightweight Lit and Unlit shaders that are already optimized for mobile platforms. Try to keep your shader variations as low as possible, as they can have a dramatic effect on runtime memory usage. If the default URP shaders don't suit your needs, you can customize the look of your materials using Shader Graph. Find out how to build your shaders visually using Shader Graph [here](#).



Create custom shaders with the Shader Graph.

## Minimize overdraw and alpha blending

Avoid drawing unnecessary transparent or semi-transparent images, and do not overlap barely visible images or effects. Mobile platforms are greatly impacted by the resulting overdraw and alpha blending. You can check overdraw using the [RenderDoc](#) graphics debugger. You can also utilize the [Rendering Debugger](#), which lets you visualize various lighting, rendering, and material properties. The visualizations help you identify rendering issues and optimize scenes and rendering configurations.



Keep post-processing effects simple in mobile applications.

## Limit post-processing effects

Fullscreen [post-processing](#) effects like glows can dramatically slow performance. Use them cautiously in your title's art direction.

## Be careful with `Renderer.material`

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.

## Optimize SkinnedMeshRenderers

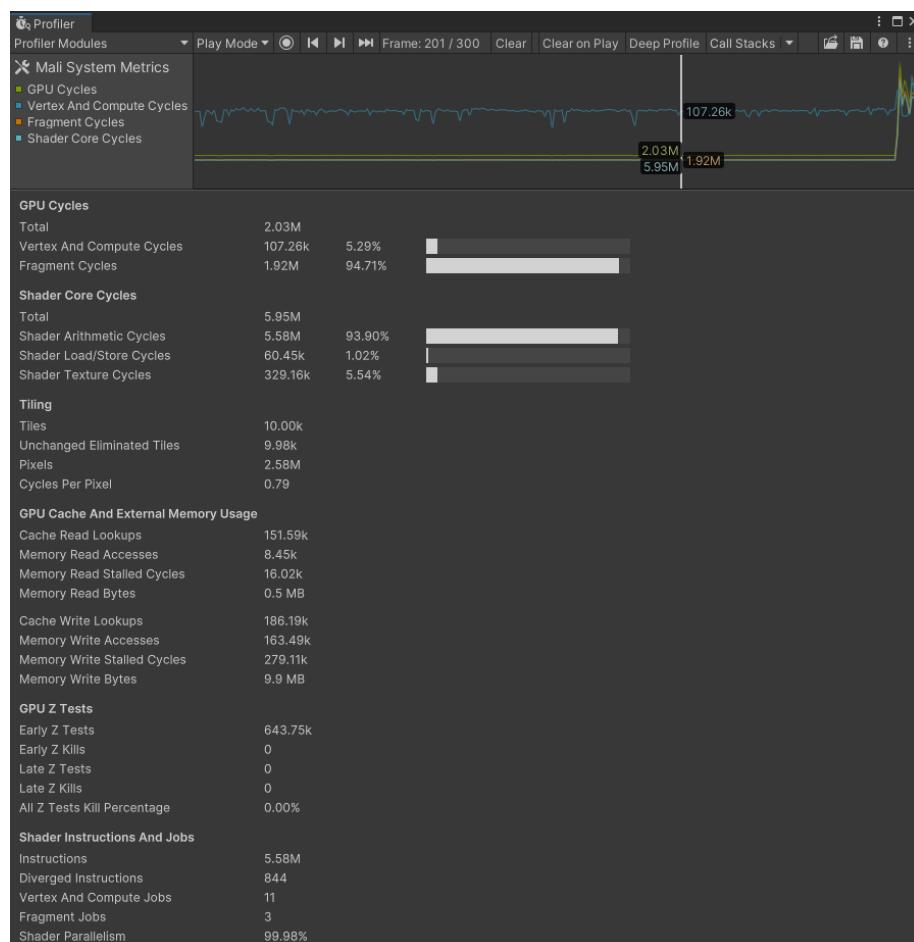
Rendering skinned meshes is expensive. Make sure that every object using a **SkinnedMeshRenderer** requires it. If a **GameObject** only needs animation some of the time, use the **BakeMesh** function to freeze the skinned mesh in a static pose, and swap to a simpler **MeshRenderer** at runtime.

## 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.

## System Metrics Mali

You can also leverage the System Metrics Mali package to access low-level system or hardware performance metrics on devices that use ARM GPUs. This includes being able to monitor low-level GPU metrics in the Unity Profiler, use the Recorder API to access low-level GPU metrics at runtime, and automate performance testing with continuous integration test runs.



Mali System Metrics Profiler Module

# USER INTERFACE

Unity offers two UI systems, the older Unity UI and the new [UI Toolkit](#). UI Toolkit is intended to become the recommended UI system. It's tailored for maximum performance and reusability, with workflows and authoring tools inspired by standard web technologies, meaning UI designers and artists will find it familiar if they already have experience designing web pages.

However, as of Unity 2022 LTS, UI Toolkit does not have some features that [Unity UI](#) and [Immediate Mode GUI \(IMGUI\)](#) support. Unity UI and IMGUI are more appropriate for certain use cases and are required to support legacy projects. See the [Comparison of UI systems in Unity](#) for more information.

## UGUI performance optimization tips

Unity UI (UGUI) is often 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, 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, potentially generating 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 keep 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

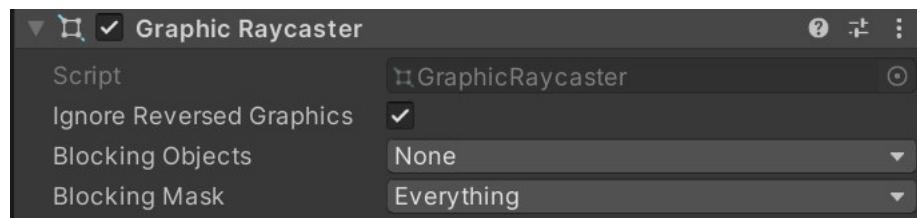
You may have UI elements that only appear sporadically in the game (e.g., a health bar that appears only when a character takes damage). If your invisible UI element is active, it might still be using draw calls. Explicitly disable any invisible UI components and re-enable them as needed.

If you only need to turn off the Canvas's visibility, disable the Canvas component rather than the whole GameObject. This can prevent your game from having to rebuild meshes and vertices when you reenable it.

### Limit GraphicRaycasters and disable Raycast Target

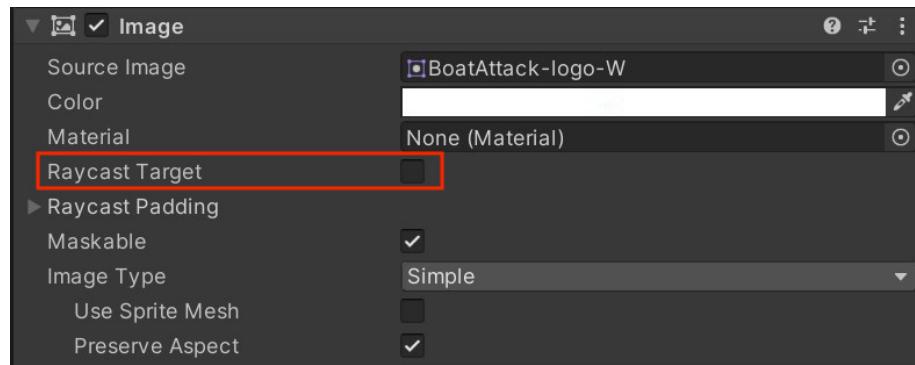
Input events like on-screen touches or clicks require the **GraphicRaycaster** component. This simply loops through each input point on screen and checks if it's within a UI's RectTransform.

Remove the default **GraphicRaycaster** from the top Canvas in the hierarchy. Instead, add the **GraphicRaycaster** exclusively to the individual elements that need to interact (buttons, scrollrects, and so on).



Disable Ignore Reversed Graphics, which is active by default.

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.

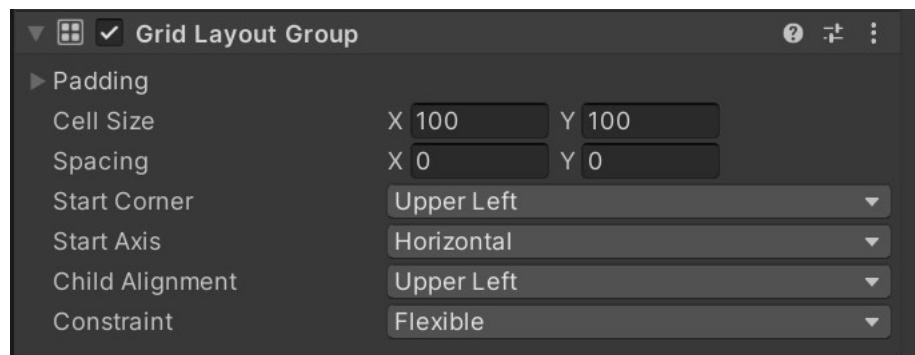


Disable Raycast Target if possible.

### 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. Alternately, 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.

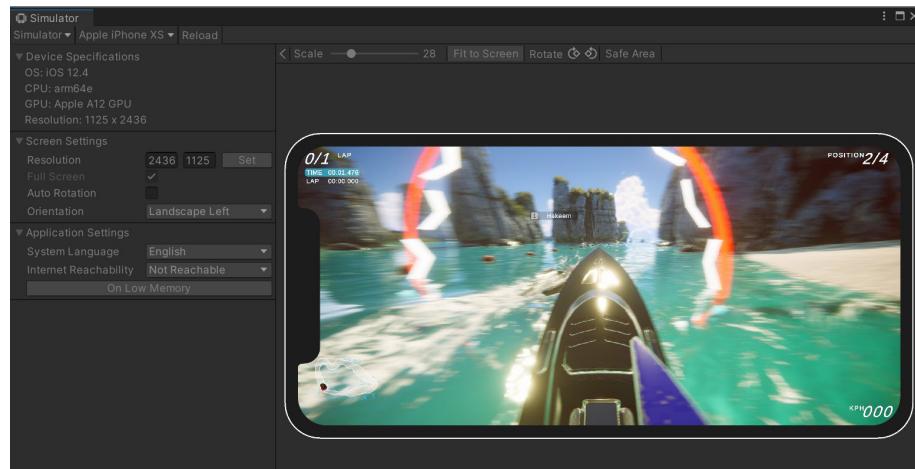
## Avoid numerous overlaid elements

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.

## Use multiple resolutions and aspect ratios

With mobile devices now using very different resolutions and screen sizes, create [alternate versions of the UI](#) to provide the best experience per device.

Use the Device Simulator to preview the UI across a wide range of supported devices. You can also create virtual devices in [XCode](#) and [Android Studio](#).



Preview a variety of screen formats using the Device Simulator.

## When using a fullscreen UI, hide everything else

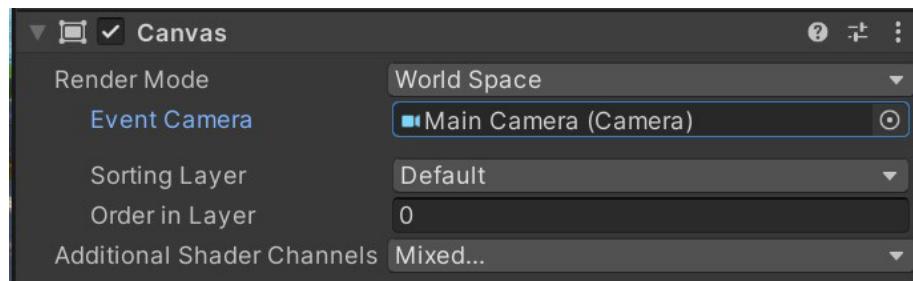
If your pause screen or start screen covers everything else in the scene, disable the camera rendering the 3D scene. Likewise, disable any background Canvas elements hidden behind the top Canvas.

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

## Assign the Camera to World Space and Camera Space Canvases

Leaving the **Event** or **Render Camera** field blank forces Unity to fill in **Camera.main**, which is unnecessarily expensive.

Consider using **Screen Space – Overlay** for your Canvas **RenderMode** if possible, since that does not require a camera.



When using World Space Render Mode, make sure to fill in the Event Camera.

## UI Toolkit performance optimization tips

UI Toolkit offers improved performance over Unity UI, is tailored for maximum performance and reusability, and provides workflows and authoring tools inspired by standard web technologies. One of its key benefits is that it uses a highly optimized rendering pipeline that is specifically designed for UI elements.

Here are some general recommendations for optimizing performance of your UI with UI Toolkit:

- **Use efficient layouts:** Efficient layouts refer to using [layout groups](#) provided by UI Toolkit, such as Flexbox, instead of manually positioning and resizing UI elements. Layout groups handle the layout calculations automatically, which can significantly improve performance. They ensure that UI elements are arranged and sized correctly based on the specified layout rules. By utilizing efficient layouts, you avoid the overhead of manual layout calculations and achieve consistent and optimized UI rendering.
- **Avoid expensive operations in Update:** Minimize the amount of work performed in Update methods, especially heavy operations like UI element creation, manipulation, or calculation. Perform these operations sparingly or during initialization when possible, since the update method is called once per frame.
- **Optimize event handling:** Be mindful of event subscriptions and unregister them when no longer needed. Excessive event handling can impact performance, so ensure you only subscribe to events that are necessary.

- **Optimize style sheets:** Be mindful of the number of style classes and selectors used in your style sheets. Large style sheets with numerous rules can impact performance. Keep your style sheets lean and avoid unnecessary complexity.
- **Profile and optimize:** Use Unity's profiling tools to identify performance bottlenecks in your UI and spot areas that can be optimized further, such as inefficient layout calculations or excessive redraws.
- **Test on target platforms:** Test your UI performance on target platforms to ensure optimal performance across different devices. Performance can vary based on hardware capabilities, so consider the target platform when optimizing your UI.

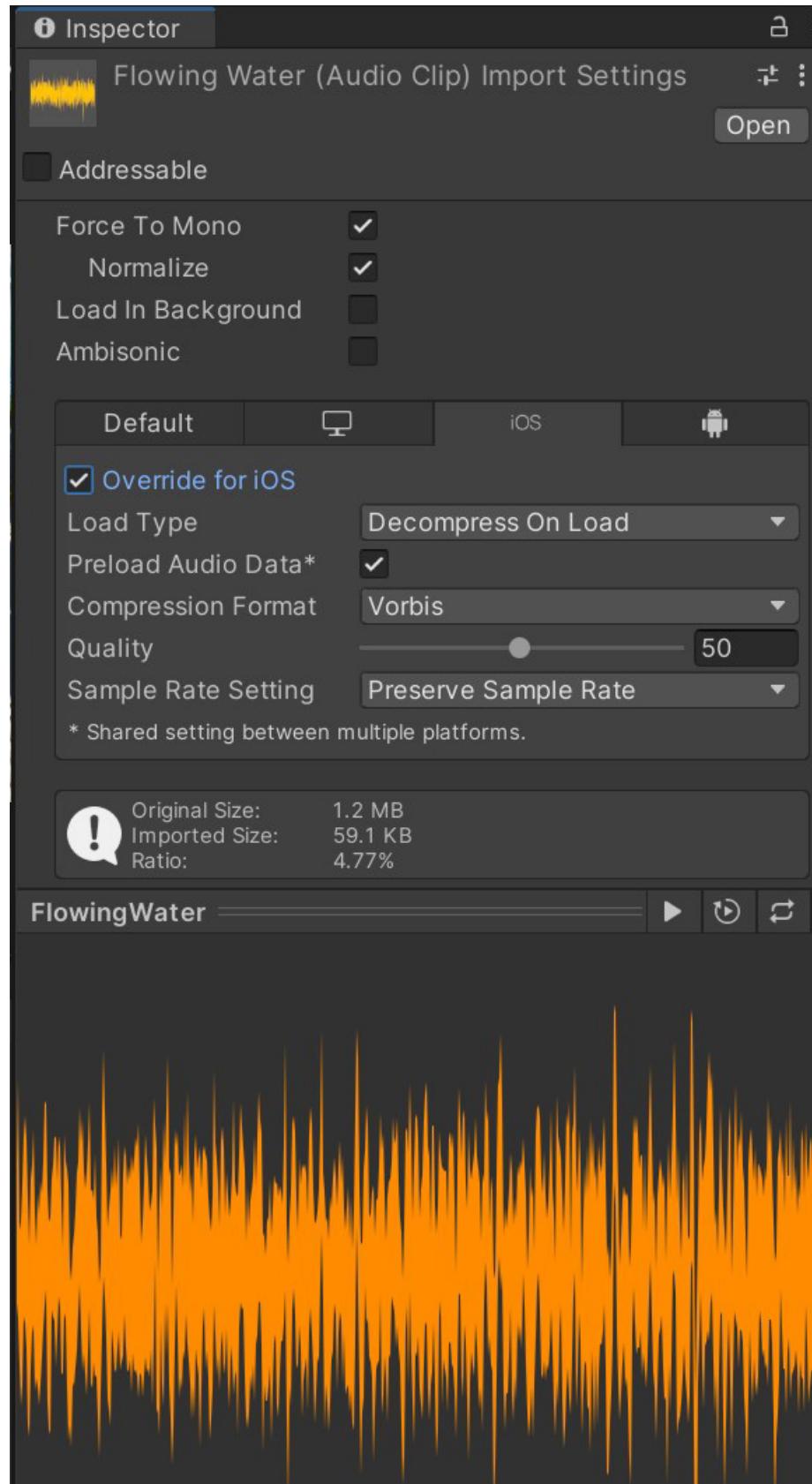
Remember, performance optimization is an iterative process. Continuously profile, measure, and optimize your UI code to ensure it runs smoothly and efficiently.

# AUDIO

Though audio is not normally a performance bottleneck, you can still optimize to save memory.

## **Make sound clips mono when possible**

If you are using 3D spatial audio, author your sound clips as mono (single channel) or enable the Force To Mono setting. A multichannel sound used positionally at runtime will be flattened to a mono source, thus increasing CPU cost and wasting memory.



Optimize the Import Settings of your AudioClips.

## **Use original uncompressed WAV files as your source assets when possible**

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.

## **Compress the clip and reduce the compression bitrate**

Reduce the size of your clips and memory usage with compression:

- Use **Vorbis** for most sounds (or **MP3** for sounds not intended to loop).
- Use **ADPCM** for short, frequently used sounds (e.g., footsteps, gunshots). This shrinks the files compared to uncompressed PCM but is fast to decode during playback.

Sound effects on mobile devices should be 22,050 Hz at most. Using lower settings usually has minimal impact on the final quality; your own ears can help you judge for yourself.

## **Choose the proper Load Type**

The setting varies per clip size.

- **Small clips (< 200 kb)** should **Decompress on Load**. This incurs CPU cost and memory by decompressing a sound into raw 16-bit PCM audio data, so it's only desirable for short sounds.
- **Medium clips (>= 200 kb)** should remain **Compressed in Memory**.
- **Large files (background music)** should be set to **Streaming**. Otherwise, the entire asset will be loaded into memory at once.

## **Unload muted AudioSources from memory**

When implementing a mute button, don't simply set the volume to 0. You can **Destroy** the  **AudioSource** component to unload it from memory, provided the player does not need to toggle this on and off very often.

# ANIMATION

Unity's [Mecanim system](#) is fairly sophisticated. If possible, limit your usage on mobile using the settings that follow.

## **Use generic versus humanoid rigs**

By default, Unity imports animated models with the Generic Rig, but developers often switch to the Humanoid Rig when animating a character.

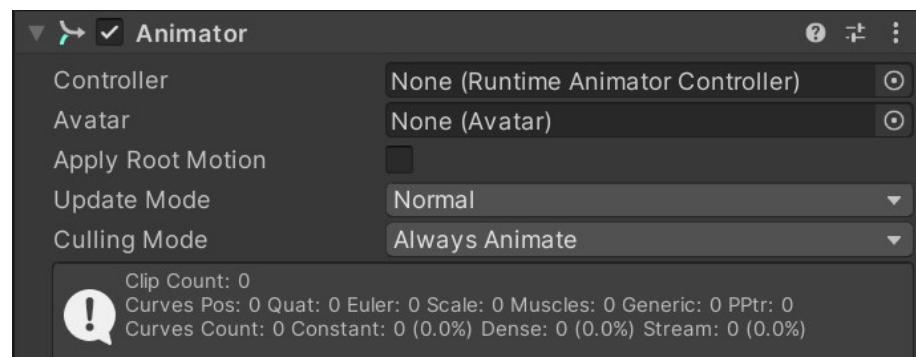
A Humanoid Rig consumes 30–50% more CPU time than the equivalent Generic Rig because it calculates inverse kinematics and animation retargeting each frame, even when not in use. If you don't need these specific features of the Humanoid Rig, use the Generic Rig instead.

## Avoid excessive use of Animators

Animators are primarily intended for humanoid characters but are often used to animate single values (e.g., the alpha channel of a UI element).

[Avoid overusing Animators](#), particularly in conjunction with UI elements. Whenever possible, use the legacy Animation components for mobile.

Consider creating tweening functions or using a third-party library for simple animations (e.g., DOTween).



Animators are potentially expensive.

# PHYSICS

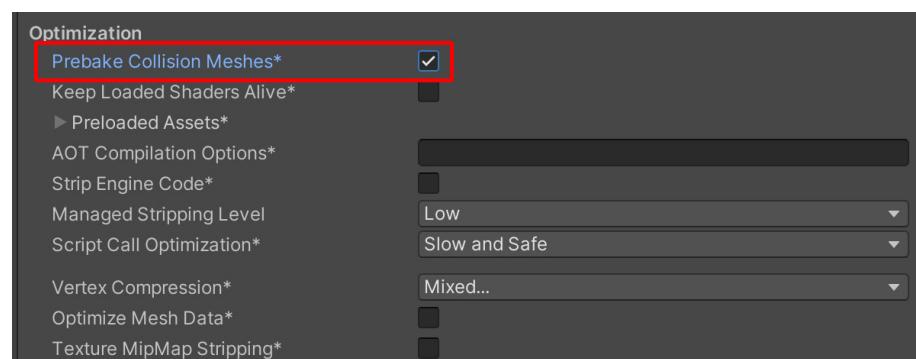
Unity's built-in Physics (Nvidia PhysX) can be expensive on mobile. The following tips may help you squeeze out more frames per second.

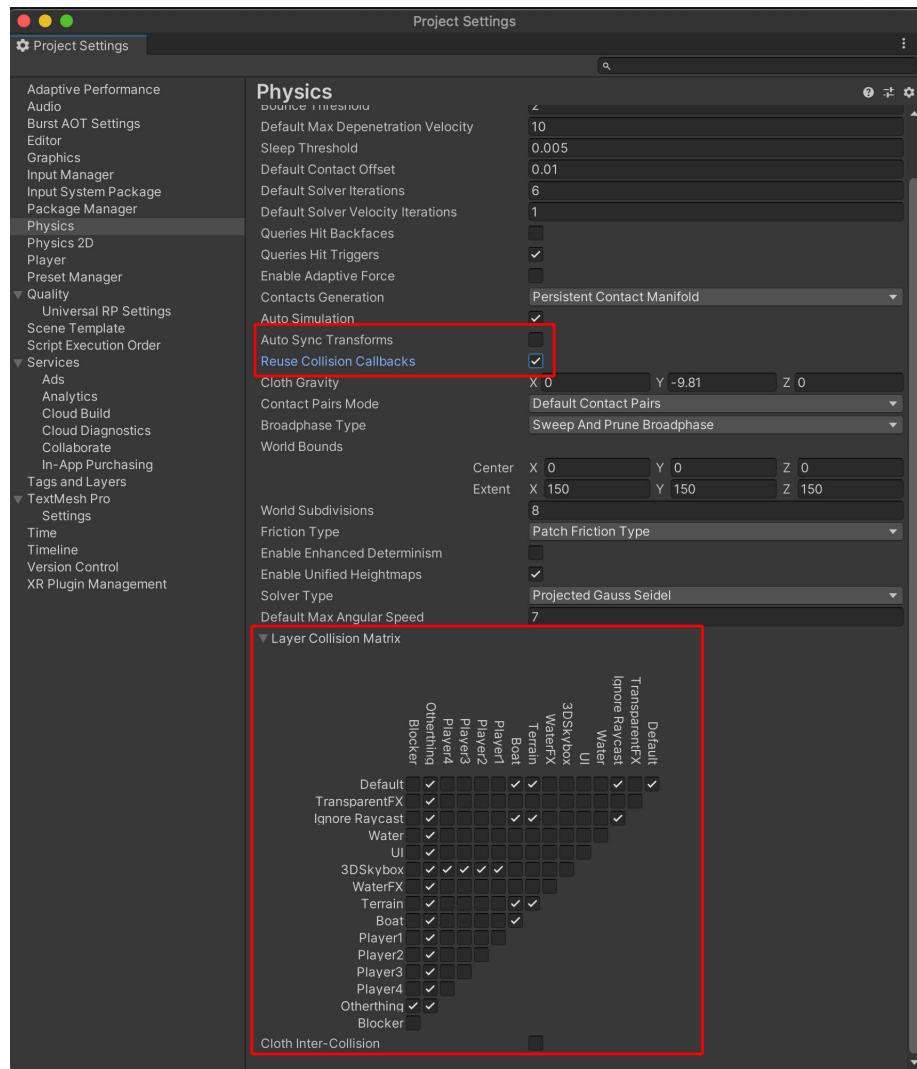
## Optimize your settings

In the [PlayerSettings](#), check **Prebake Collision Meshes** whenever possible.

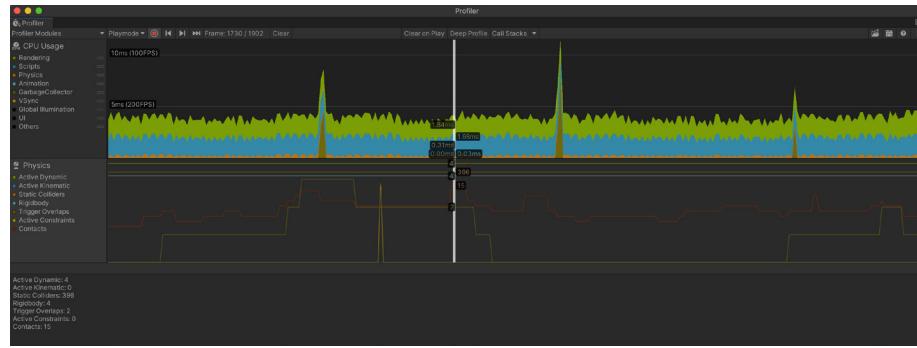
Make sure that you edit your **Physics settings** ([Project Settings > Physics](#)) as well. Simplify your **Layer Collision Matrix** wherever possible.

Disable **Auto Sync Transforms** and enable **Reuse Collision Callbacks**.





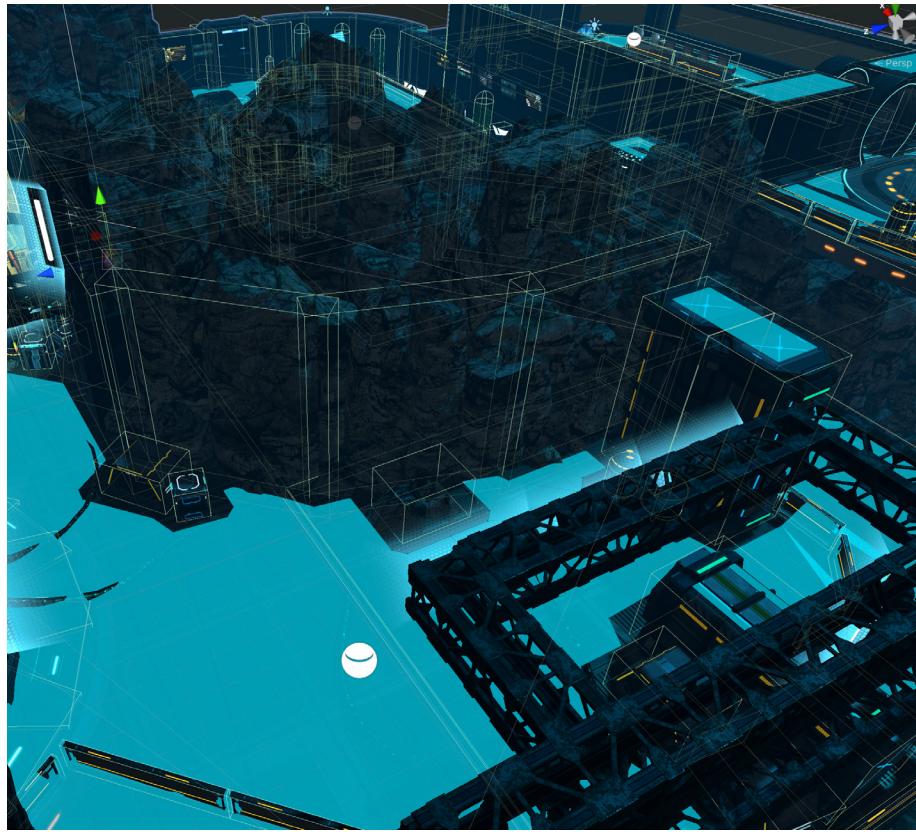
Modify the physics project settings to squeeze out more performance.



Keep an eye on the [Physics](#) module of the Profiler for performance issues.

## Simplify colliders

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



Use primitives or simplified meshes for colliders.

## Move a Rigidbody using physics methods

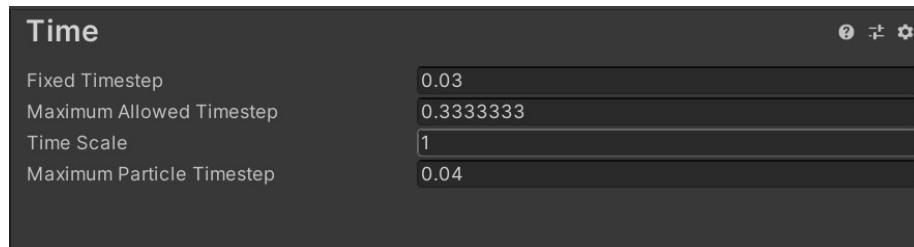
Use class methods like **MovePosition** or **AddForce** to move your **Rigidbody** objects. Translating their **Transform** components directly can lead to physics world recalculations, which can be expensive in complex scenes. Move physics bodies in **FixedUpdate** rather than **Update**.

### Fix the Fixed Timestep

The default **Fixed Timestep** in the Project Settings is 0.02 (50 Hz). Change this to match your target frame rate (for example 0.03 for 30 fps).

Otherwise, if your frame rate drops at runtime, that means Unity would call **FixedUpdate** multiple times per frame, potentially creating a CPU performance issue with physics-heavy content.

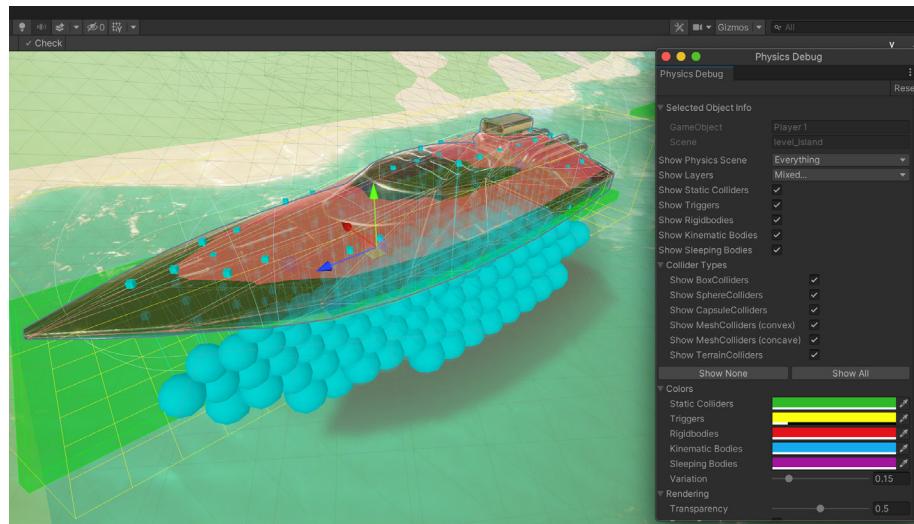
The **Maximum Allowed Timestep** limits how much time physics calculations and FixedUpdate events can use in the event the frame rate drops. Lowering this value means that physics and animation can slow down, while also reducing their impact on the frame rate during a performance hitch.



Modify the Fixed Timestep to match your target frame rate, and lower the Maximum Allowed Timestep to reduce performance glitches.

### 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 what GameObjects should be able to collide with one another.



The Physics Debugger helps you visualize how your physics objects can interact with each other.

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

# WORKFLOW AND COLLABORATION

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

## Use version control

Everyone should be using some type of version control. Make sure your **Editor Settings** have **Asset Serialization Mode** set to **Force Text**.



If you're using an external version control system (such as Git) in the **Version Control** settings, make sure the **Mode** is set to **Visible Meta Files**.



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

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.

For further versioning support, check out [Plastic SCM](#), our recommended version control solution for Unity game development.

### **Break up large Scenes**

Large, single Unity scenes do not lend themselves well to collaboration. Divide your levels into multiple smaller scenes so 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.

### **Remove unused resources**

Watch out for any unused assets that come bundled with third-party plug-ins and libraries. Many include embedded test assets and scripts, which will become part of your build if you don't remove them. Strip out any unneeded resources left over from prototyping.

# Reach the next level with industry-leading expertise from Accelerate Solutions

Accelerate Solutions specializes in helping game studios hit their most ambitious goals across several use cases, including: improving performance and optimization, game planning and technical design, project acceleration, improving player KPIs and monetization, and delivering on challenging ports and migrations. The global team is made up of Unity's most senior software developers and technical artists, with hands-on knowledge across the Unity engine, multiplayer, cloud, devops, AI/ML, and game design.

The team's expertise lies in helping you take your game to the next level, no matter what stage of game development you're in. Mainly, the optimizations focus on identifying general and specific performance issues such as frame rate, memory, and binary size to improve player experiences and/or iteration times.

Services offered range from consulting to full game development.

- **Consulting**

During these engagements, the consultant will analyze your project or workflows and provide guidance and recommendations to your team on how to achieve your desired outcome.

- **Codevelopment**

Working alongside your team, a Unity developer and/or team will deep dive into your project and achieve a desired outcome.

- **Custom development**

For these engagements, the Accelerate Solutions team will assign and partner with an internal Unity team that will lead and execute a project on your behalf, owning it from inception to completion.

- **Full game development**

As the name implies, these are engagements where Accelerate Solutions assigns a highly experienced Unity game studio team that will lead and execute a project on your behalf, owning it from inception to completion.

To learn more about Accelerate Solutions, please [reach out](#).

# Remove roadblocks with Unity Integrated Success

From strategic planning to unforeseen challenges, if you need personalized attention, consider [Unity Integrated Success](#). Integrated Success is our most complete Success Plan for your most complex projects. Get insight, hands-on guidance, and premium technical support to ensure your project's success. Guaranteed response times and prioritized bug handling allow you to quickly overcome roadblocks.

Integrated Success also allows you to optionally add read and modification access to Unity source code. This access is available for development teams that want to deep dive into Unity source code to adapt and reuse it for other applications.

## Optimize your game with a Project Review

Project Reviews are an essential part of the Integrated Success package. Learn how to optimize your project during this annual review. Senior engineers perform an analysis of your work and provide insights and actionable advice specific to your goals. The team familiarizes themselves with your projects and then uses various profiling tools to detect performance bottlenecks, factoring in existing requirements and design decisions. They also try 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.), they'll make adjustments and reprofile to uncover new issues. In instances where the team is unable to solve problems immediately, they'll capture as much information as possible and conduct further investigation internally, consulting specialized developers across R&D if necessary.

Though deliverables can vary depending on your needs, findings are summarized with recommendations provided in a written report. The team's goal is to always provide the greatest value to you by helping to identify potential blockers, assess risk, validate solutions, and ensure that best practices are followed moving forward.

## Partner Relations Manager (PRM)

In addition to a Project Review, Unity Integrated Success also comes with a Partner Relations Manager (PRM) – a strategic Unity advisor who acts as your internal advocate and an extension of your team to help you get the most out of Unity. They maintain clear lines of communication so you're always informed and working towards your goals. Your PRM provides you with the dedicated technical and operational expertise required to preempt issues and keep your projects running smoothly, up to and following launch.

To learn more about our Integrated Success packages, Project Reviews, and PRMs, please [reach out](#).

# CONCLUSION

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 mobile 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.

Of course, a little bit of creativity helps here, too.

## More resources

[Create a C# style guide: Write cleaner code that scales](#) assists you with developing a style guide to help unify your approach to creating a more cohesive codebase.

[Level up your code with game programming patterns](#) highlights best practices for using the SOLID principles and common programming patterns to create scalable game code architecture in your Unity project.

[Create modular game architecture in Unity with ScriptableObjects](#) provides tips and tricks from professional developers for deploying ScriptableObjects in production. These include examples showing how to apply them to specific design patterns and how to avoid common pitfalls.

## Professional training for Unity creators

Unity Professional Training gives you the skills and knowledge to work more productively and collaborate efficiently in Unity. Find an extensive training catalog designed for professionals in any industry, at any skill level, in multiple delivery formats.

All materials are created by experienced Instructional Designers in partnership with our engineers and product teams. This means that you always receive the most up-to-date training on the latest Unity tech.

[Learn more](#) about how Unity Professional Training can support you and your team.



[unity.com](https://unity.com)