UNITY FOR GAMES

E-BOOK

OPTIMIZE YOUR GAME PERFORMANCE FOR MOBILE

2020 LTS EDITION

Unity

Contents

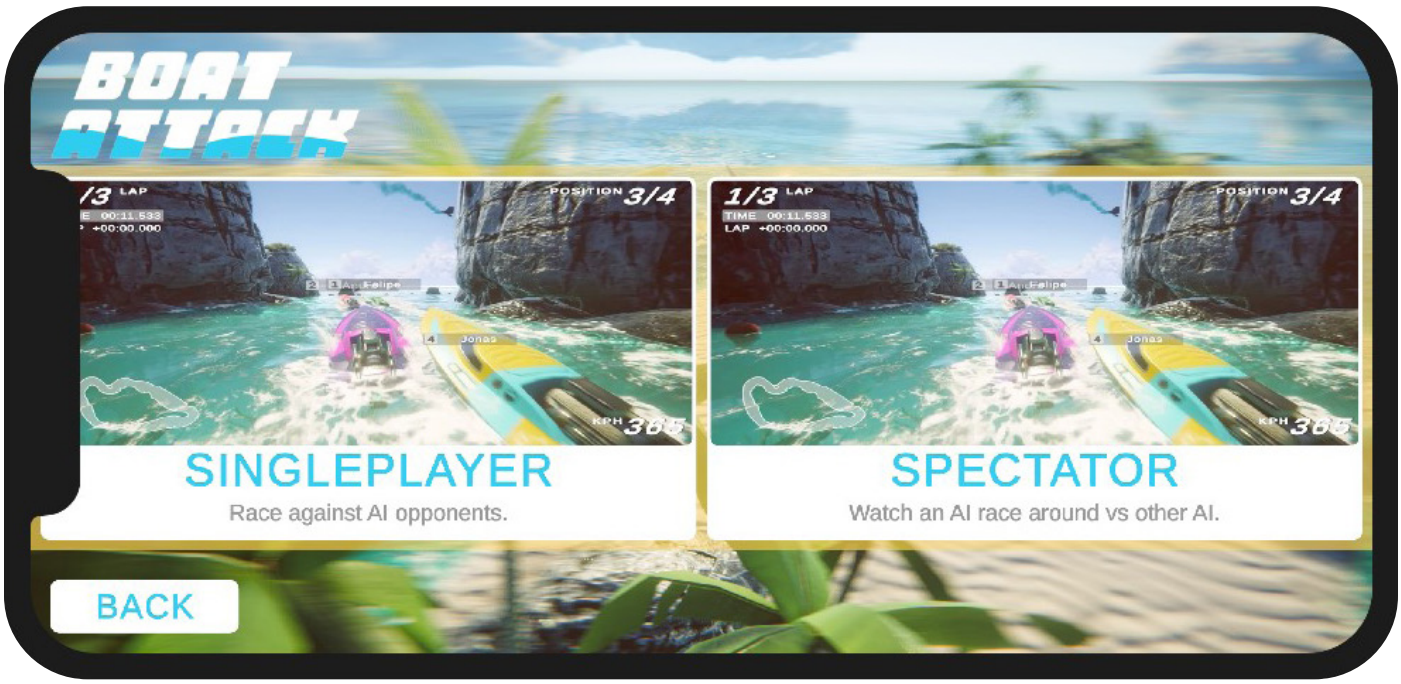
Introduction

Making mobile games is hard, and ensuring a quality experience across a fragmented device ecosystem can be even harder. Optimizing your mobile applications is an essential process that underpins the entire game 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 reaching the *billions*. If your mobile game is highly optimized, it has a better chance at passing certification from platform-specific stores and reaching a larger audience. To maximize your opportunity for success at launch and beyond warrants the creation of a deeply immersive experience that is performant on the widest range of handhelds.

This guide assembles knowledge and advice from Unity's expert team of software engineers and their experience building tools that have been used to launch hundreds of thousands of mobile games. Follow the steps outlined here to enhance your mobile game's performance while reducing its power consumption.

Start optimizing your mobile game with support from the Unity team.[1]



1 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

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.

Along with the Unity Profiler, you can leverage native tools from iOS and Android for further performance testing on their respective engines:

— On iOS, use Xcode and Instruments.

— On Android, use Android Studio and Android Profiler.

Certain hardware can take advantage of additional profiling tools (e.g., Arm Mobile Studio, Intel VTune, and Snapdragon Profiler). 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.

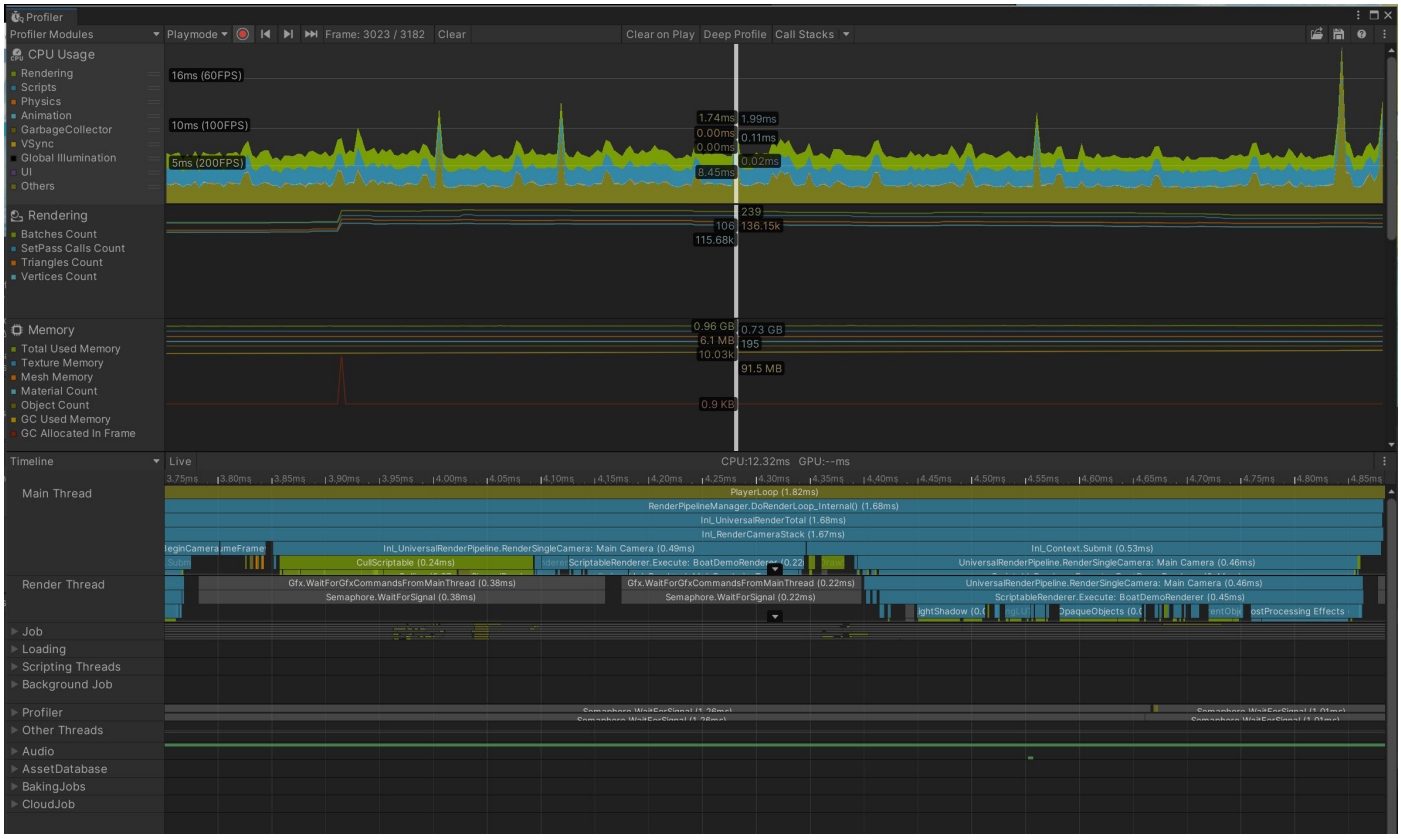
Of course, not every optimization described here will apply to your application. Something that works well in one project may not translate to yours. Identify genuine bottlenecks and concentrate your efforts on what benefits your work.

Understand how the Unity Profiler works

The Unity Profiler can help you detect the causes of any lags or freezes at runtime and better understand what's happening at a specific frame, or point in time.

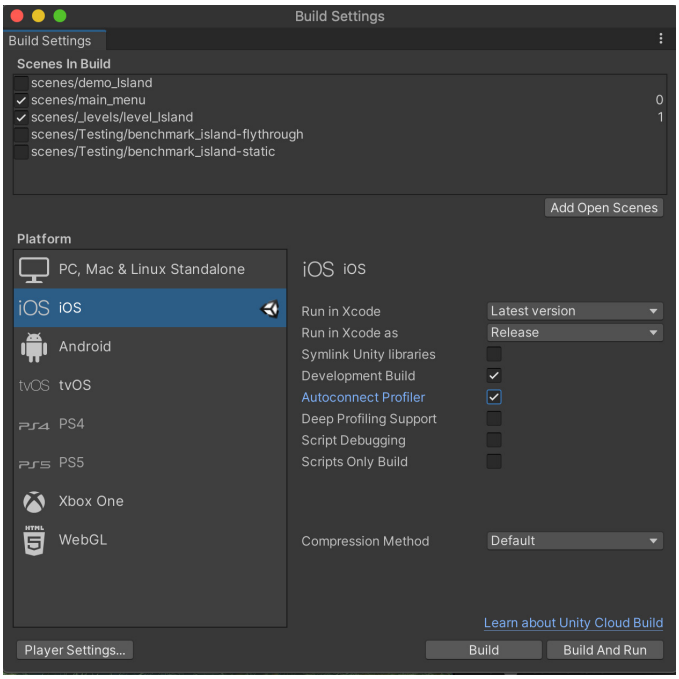
It's an instrumentation-based profiler that profiles code timings explicitly wrapped in ProfileMarkers (such as MonoBehaviour's Start or Update methods, or specific API calls).

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

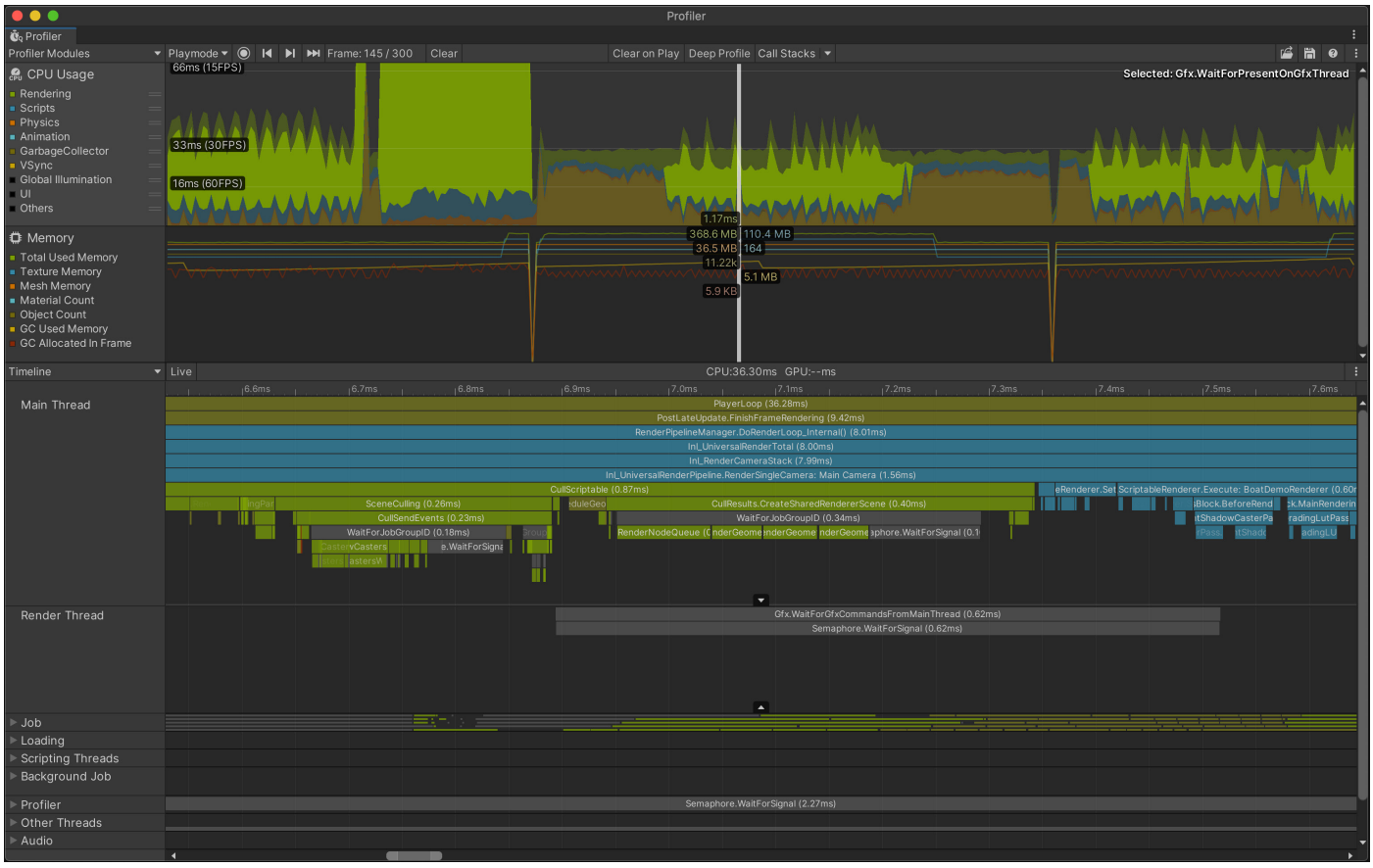


Use the Unity Profiler to test performance and resource allocation for your application.

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.



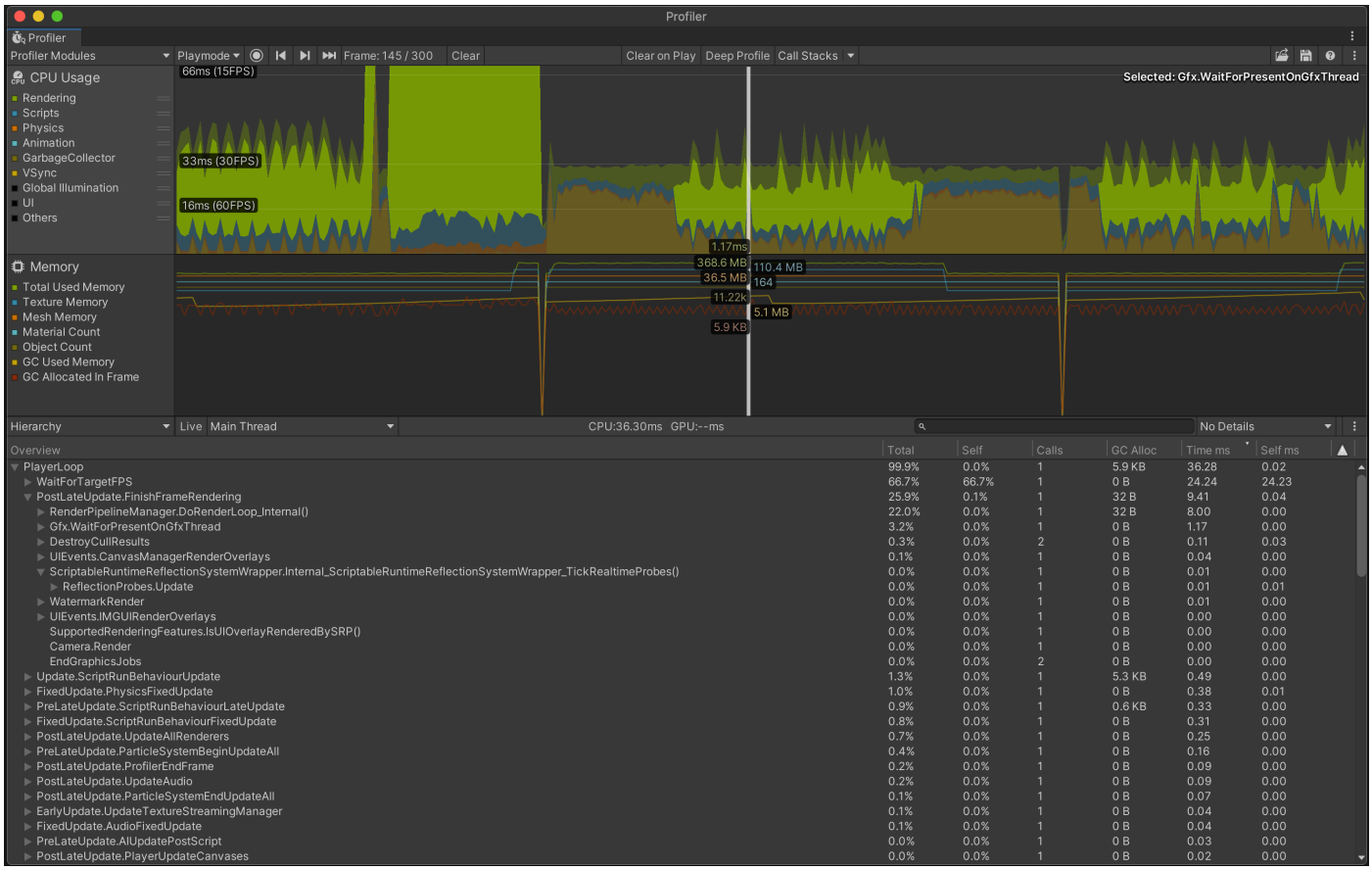
Use the Timeline view to determine if you are CPU-bound or GPU-bound.

When using the Deep Profiling setting, Unity can profile the beginning and end of every function call in your script code to tell you exactly which part of your application is causing a slowdown.

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.



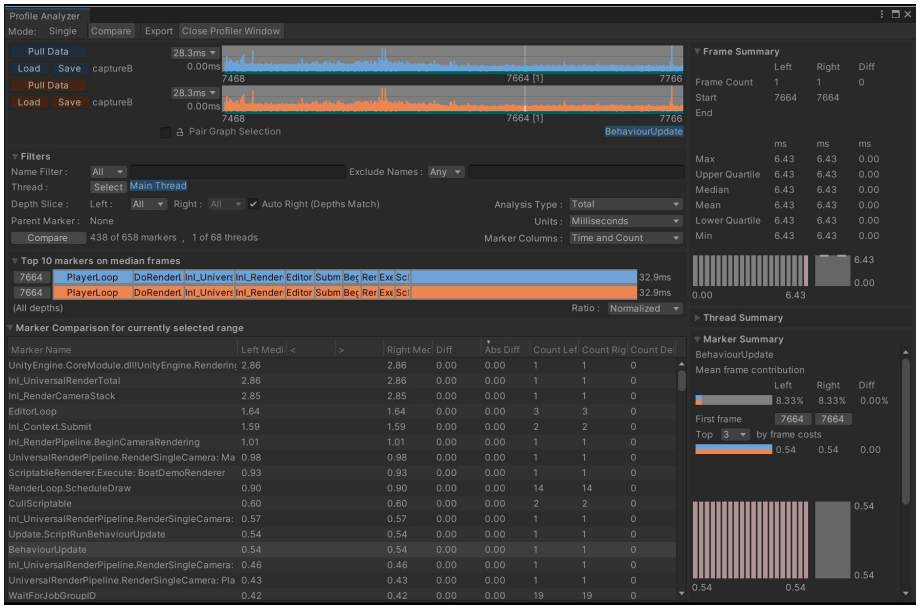
The Hierarchy view allows you to sort ProfileMarkers by time cost.

Read a complete overview of the Unity Profiler here. Those new to profiling can also watch this 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

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 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). Ideally, an application running at 30 fps will allow for approximately 33.33 ms per frame (1000 ms / 30 fps). Likewise, a target of 60 fps leaves 16.66 ms per frame.

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

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.

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 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.WaitForPresent**, it means that the main thread is ready but might be 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.

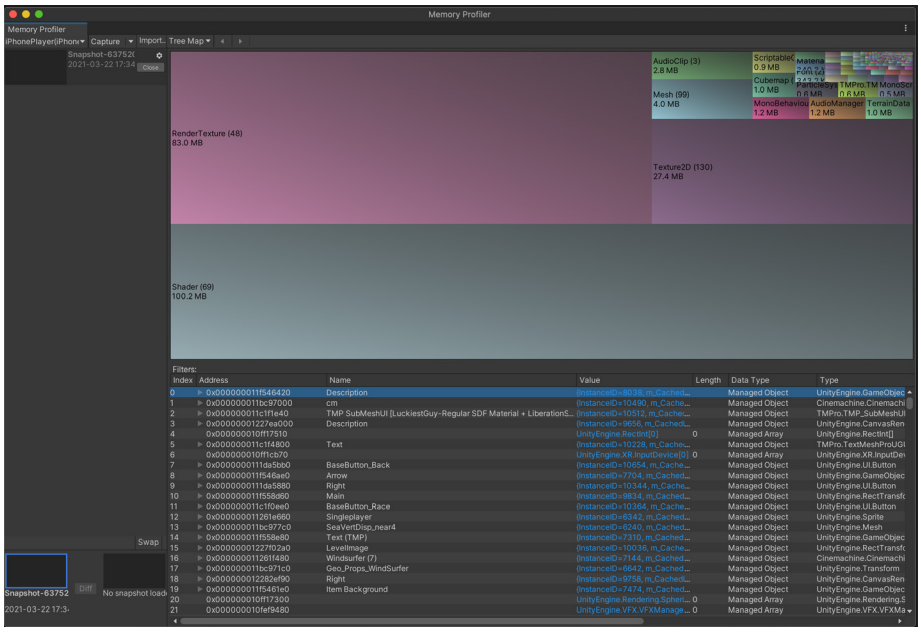
Memory

Unity employs automatic memory management for your user-generated code and scripts. Small pieces of data, like value-typed local variables, are allocated 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 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 an Experimental or Preview package in the Package Manager) can take a snapshot of your managed heap 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. You can also check out our official Memory Profiler documentation.

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 that can 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 like MessagePack or Protobuf instead. Use the StringBuilder class if you need to build strings at runtime.

— **Unity function calls:** 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** (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.

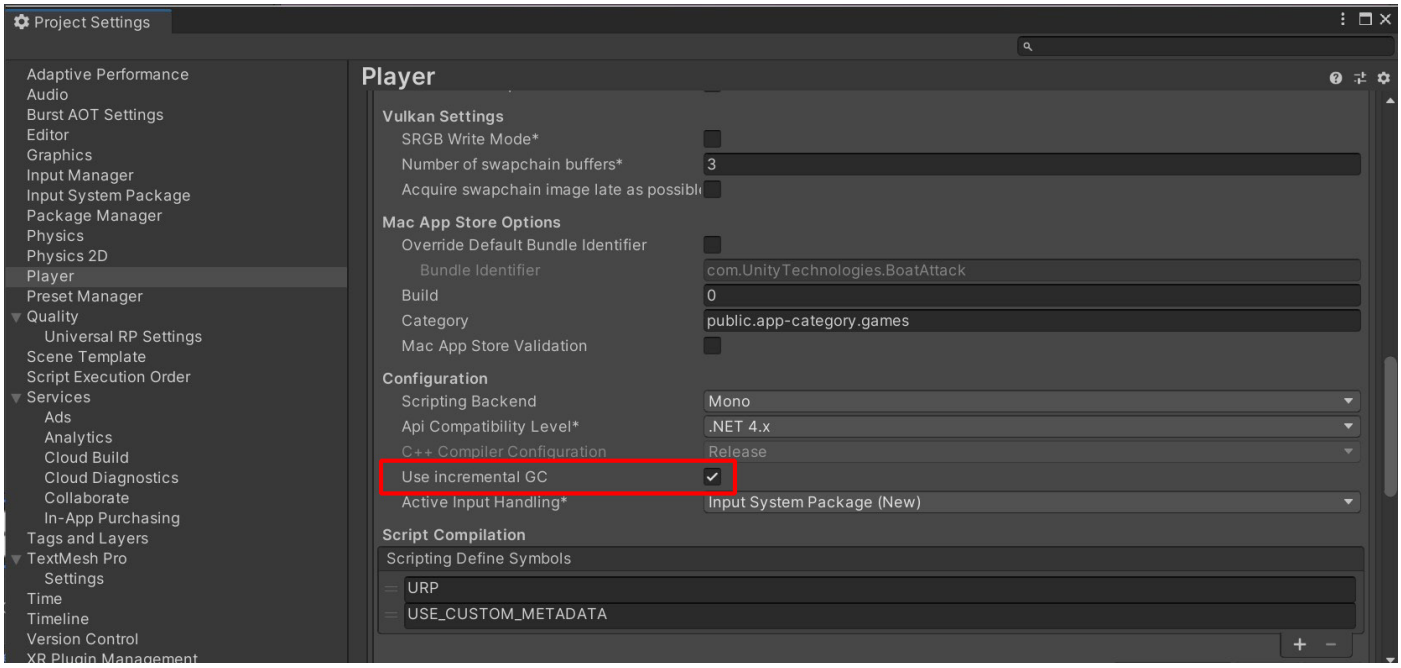
Time garbage collection whenever possible

If you are certain that a garbage collection freeze won't affect a specific point in your game, you can trigger garbage collection with **System.GC.Collect**.

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

Use the Incremental Garbage Collector to split the GC workload

Rather than creating a single, long interruption during your program's execution, incremental garbage collection uses multiple, 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.[2]



Use the Incremental Garbage Collector to reduce GC spikes.

2 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.

Adaptive Performance

With Unity and Samsung's Adaptive Performance, you can monitor the device's thermal and power state to ensure that you are ready to react appropriately. When users play for an extended period of time, you can reduce your level of detail (LOD) bias dynamically so that your game continues 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, Adaptive Performance also offers several automatic modes. In these modes, Adaptive Performance determines the game settings along several key metrics:

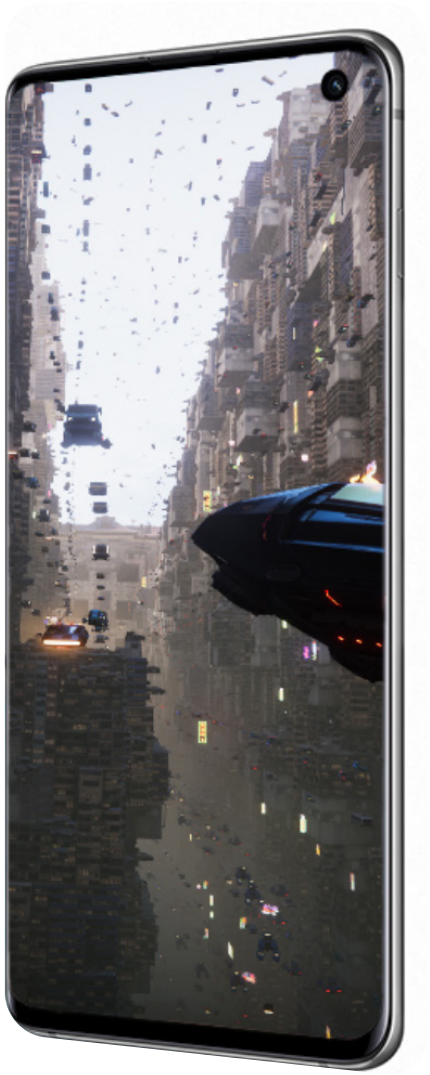
— Desired frame rate based on previous frames

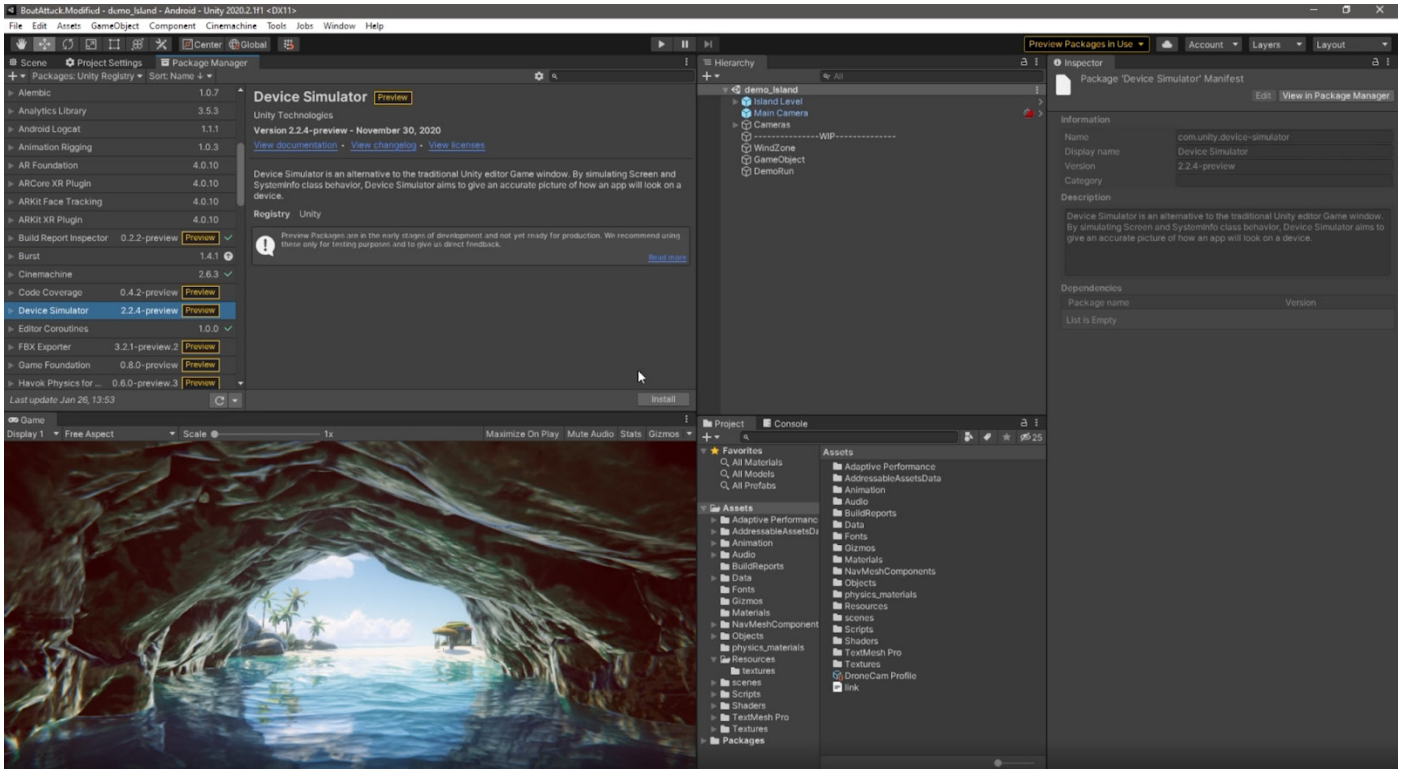
— Device temperature level

— Device proximity to thermal event

— Device bound by CPU or GPU

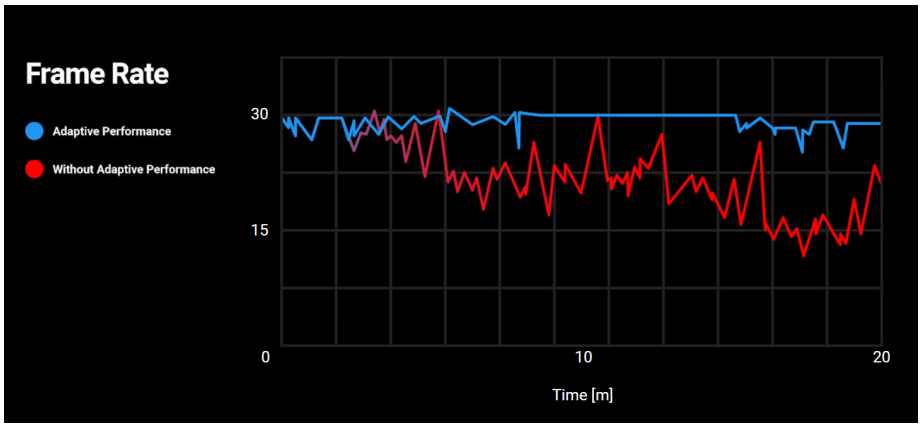
These four metrics dictate the state of the device, and Adaptive Performance tweaks the adjusted settings to reduce the bottleneck. This is done by providing an integer value, known as an Indexer, to describe the state of the device.





Note that Adaptive Performance only works for Samsung devices.

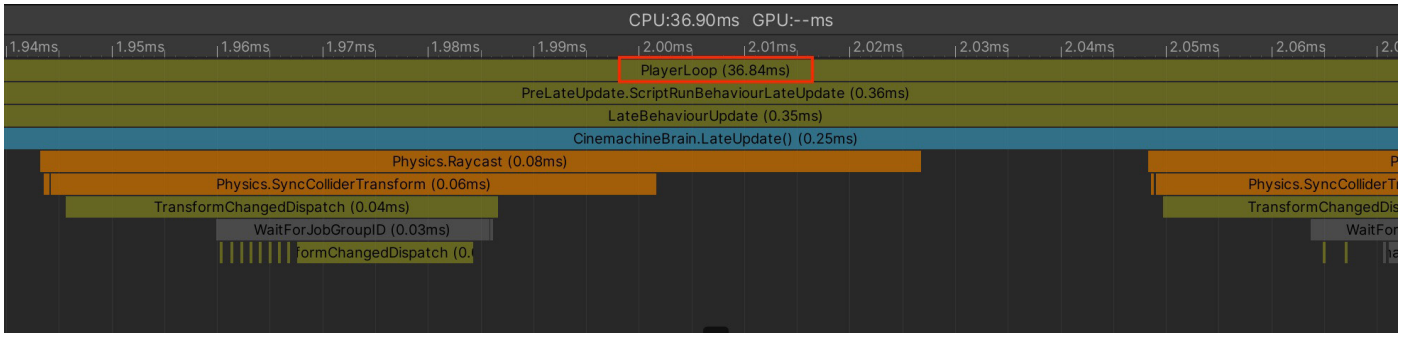
To learn more about Adaptive Performance, you can view the samples we've provided in the Package Manager by selecting **Package Manager > Adaptive Performance > Samples**. Each sample interacts with a specific scaler, so you can see how the different scalers impact your game.
We also recommend reviewing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



Programming and code architecture

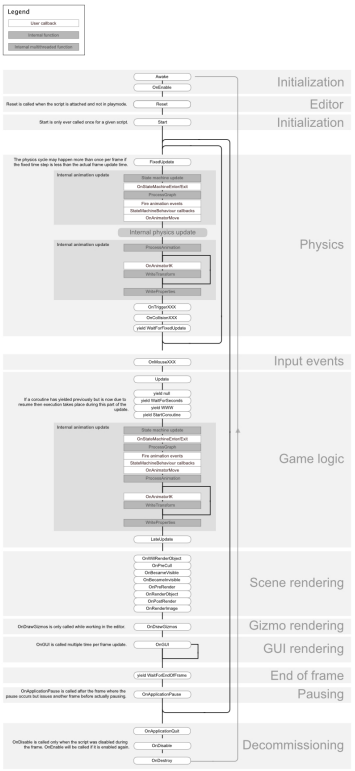
The Unity PlayerLoop contains functions for interacting with the core of the game engine. This structure includes a number of systems that handle initialization and per-frame updates. All of your scripts will rely on this PlayerLoop to create gameplay.

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.

You can optimize your scripts with the following tips and tricks.



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

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.

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();
  }
}
```

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 might encounter longer loading times than necessary.

Refer to the order of execution for event functions for details on 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 employing these methods for testing:

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

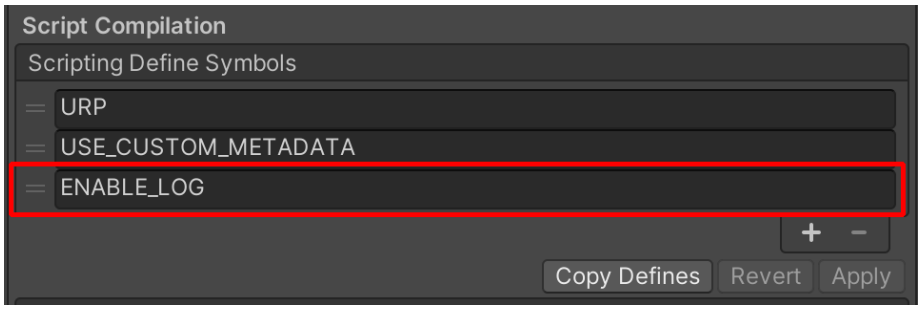
Here, you can freely use the Update in-Editor for testing without unnecessary overhead slipping into your build.

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.

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 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 it's best to avoid calling them in Update methods. Instead, call them in Start and cache the results.

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

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

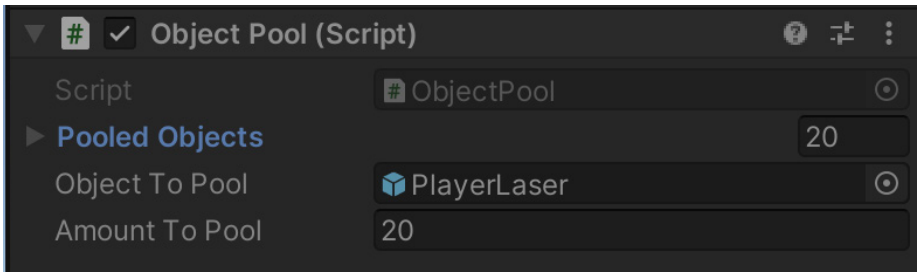
It's more efficient to invoke **GetComponent** only once, as the result of the function is cached. The cached result can be reused in Update without any further calls to GetComponent.

```
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, which generally results in 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.

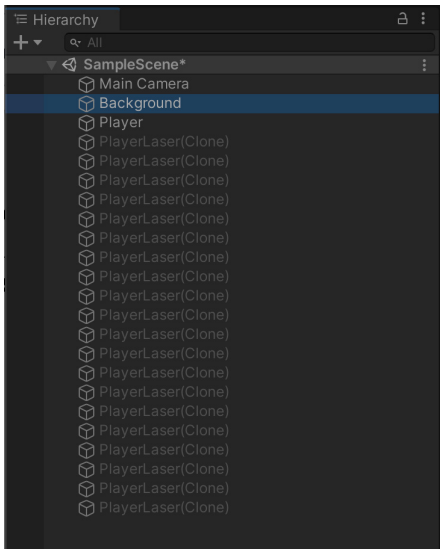


In this example, the ObjectPool creates 20 PlayerLaser instances for reuse.

Create the reusable instances when a CPU spike is less noticeable (e.g., during a menu screen). 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.

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.

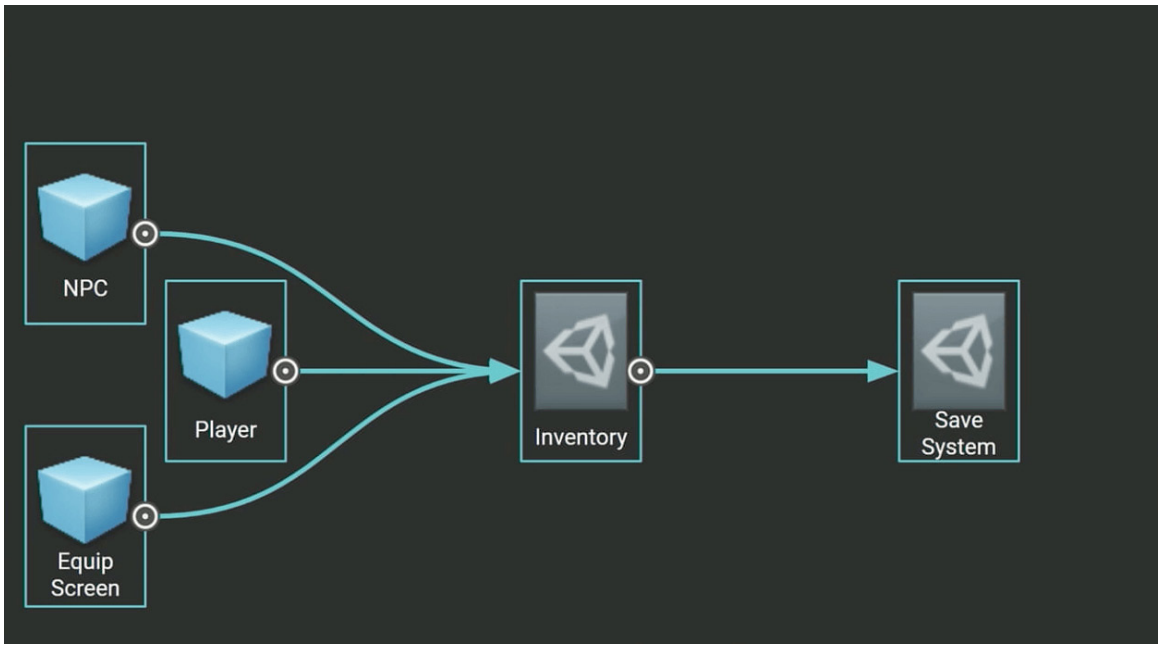


The pool of PlayerLaser objects is inactive and ready to shoot.

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

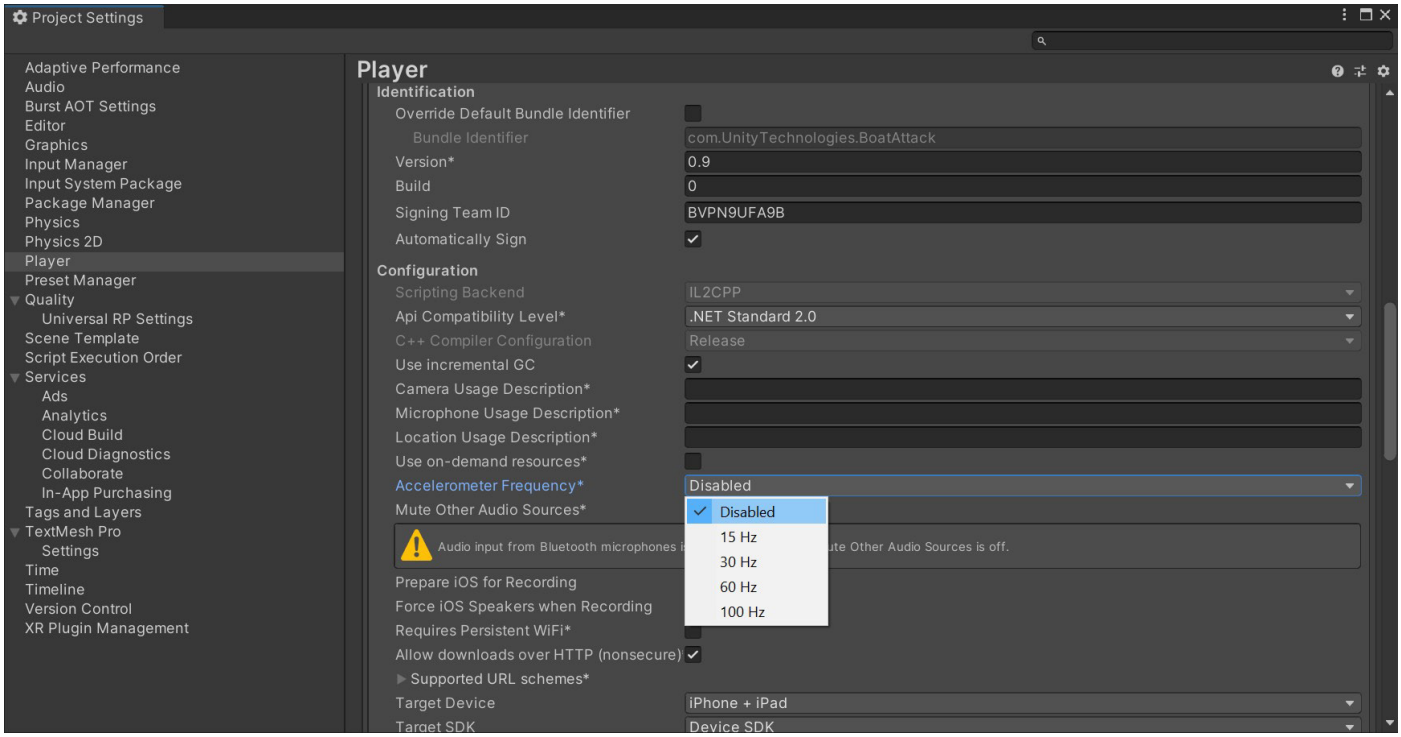
Watch this Introduction to ScriptableObjects tutorial to see how ScriptableObjects can benefit your project. You can also find relevant documentation here.

Project configuration

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

Reduce or disable Accelerometer Frequency

Unity pools your mobile's accelerometer several times per second. Disable this if it's not being used in your application, or reduce its frequency for better overall 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 needless 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 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 and this Unite talk for best practices with Transforms.

Transform once, not twice

Additionally, 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.

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 for smooth processes.

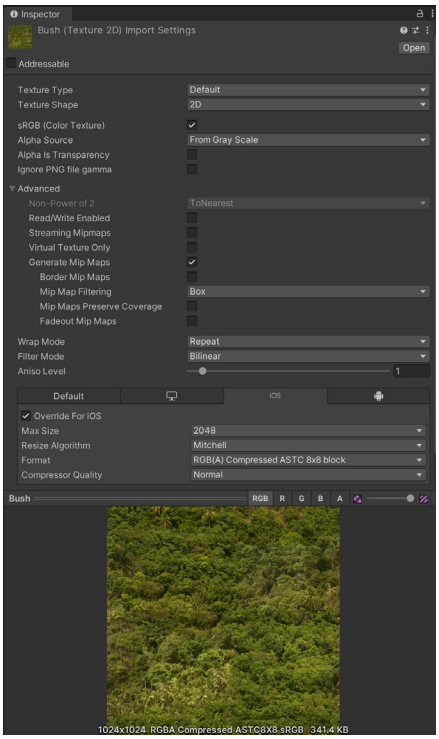
Don't rely on default settings. Use the platform-specific override tab to optimize assets such as textures and mesh geometry. Incorrect settings might yield larger build sizes, longer build times, and poor memory usage. Consider using the Presets feature to help customize baseline settings that will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

See this guide on best practices for art assets or check out this course on 3D Art Optimization for Mobile Applications via 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, try to follow these guidelines:

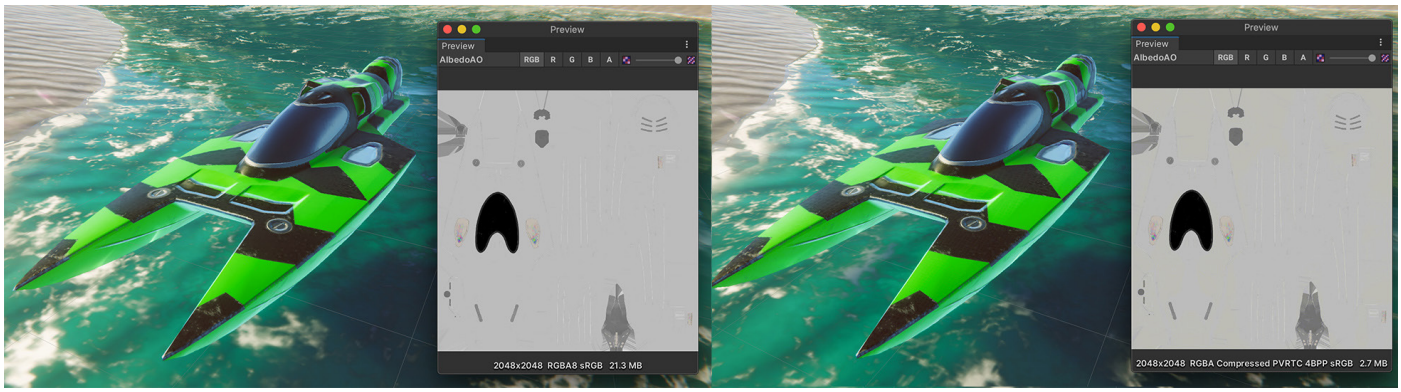
- **Lower the Max Size:** Use the minimum settings that produce visually acceptable results. This is non-destructive and can quickly reduce your texture memory.
- **Use powers of two (POT):** Unity requires POT texture dimensions for mobile texture compression formats (PVRCT or ETC).
- **Atlas your textures:** Placing multiple textures into a single texture can reduce draw calls and speed up rendering. Use the Unity Sprite Atlas or the third-party TexturePacker 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 onscreen, such as 2D sprites and UI graphics (leave Mip Maps enabled for 3D models with varying 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 improvement in visual quality.



Uncompressed textures require more memory.

Use Adaptive Scalable Texture Compression (ATSC) for both iOS and Android. The vast majority of games in development tend to target min-spec devices that support ATSC 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 compressed formats such as PVRTC and ETC aren't sufficiently high-quality, 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

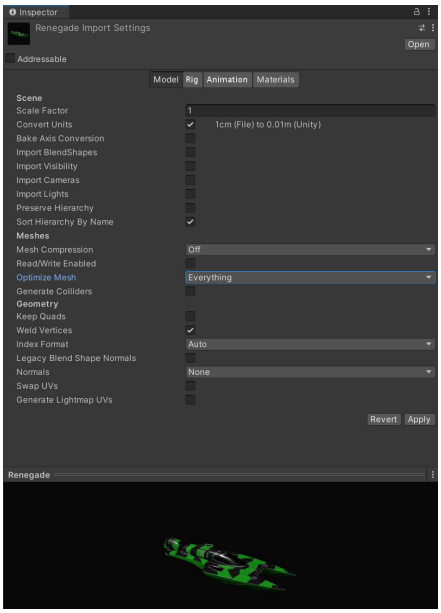
Much like textures, meshes can consume excess memory if not imported carefully. To minimize meshes' memory consumption:

— **Compress the mesh:** Aggressive compression can reduce disk space (memory at runtime, however, is unaffected). Note that mesh quantization can result in inaccuracies, 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 that the mesh material will not need normals or tangents, uncheck these options for extra savings.



Check your mesh import settings.

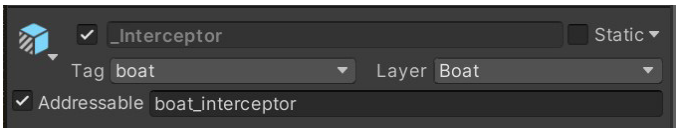
Check your polygon counts

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, and 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 run scripts when importing assets. This way, you can customize settings before and/or after importing models, textures, audio, etc.

Use the Addressable Asset System

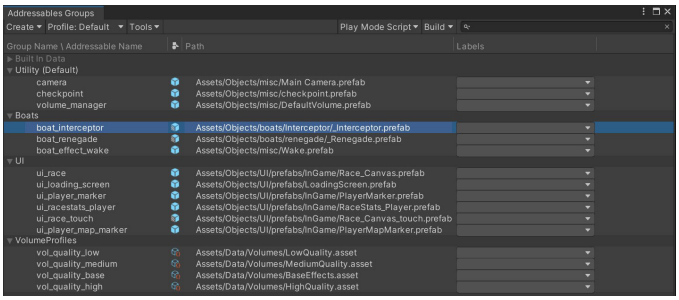


The Addressable Asset System provides a simplified way to manage your content. This unified system loads AssetBundles by "address" or alias, asynchronously from either a local path or a remote content delivery network (CDN).

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.

Click here to see how the Addressable Asset System can take the hassle out of asset management.



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

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) currently uses a single-pass forward renderer to bring high-quality graphics to your mobile platform (deferred rendering will be available in future releases). The same physically based Lighting and Materials from consoles and PCs can also scale to your phone or tablet.

The following guidelines can help speed up your graphics.

Batch your draw calls

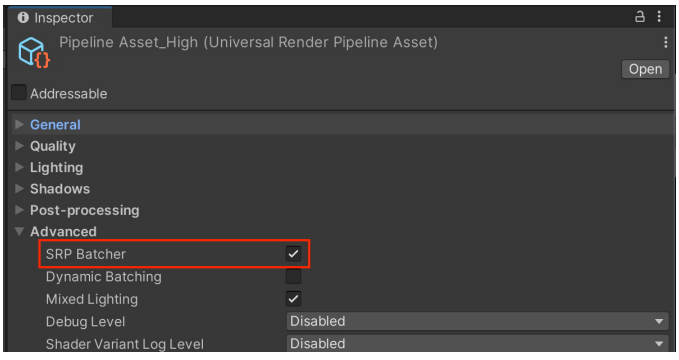
Batching objects to be drawn together minimizes the state changes needed to draw each object in a batch. This leads to improved performance by reducing the CPU cost of rendering objects. Unity can combine multiple objects into fewer batches using several techniques:

— **Dynamic batching:** For small meshes, Unity can group and transform vertices on the CPU, then draw them all in one go. Note: Only use this if you have enough low-poly meshes (less than 900 vertex attributes and no more than 300 vertices). The Dynamic Batcher won't batch meshes larger than this, so enabling it will waste CPU time looking for small meshes to batch in every frame.

— **Static batching:** For non-moving geometry, Unity can reduce draw calls for meshes that share the same material. While it is more efficient than dynamic batching, it uses more memory.

— **GPU instancing:** If you have a large number of identical objects, this technique batches them more efficiently through the use of graphics hardware.

— **SRP Batching:** Enable the SRP Batcher in your Universal Render Pipeline Asset under Advanced. This can speed up your CPU rendering times significantly, depending on the Scene.

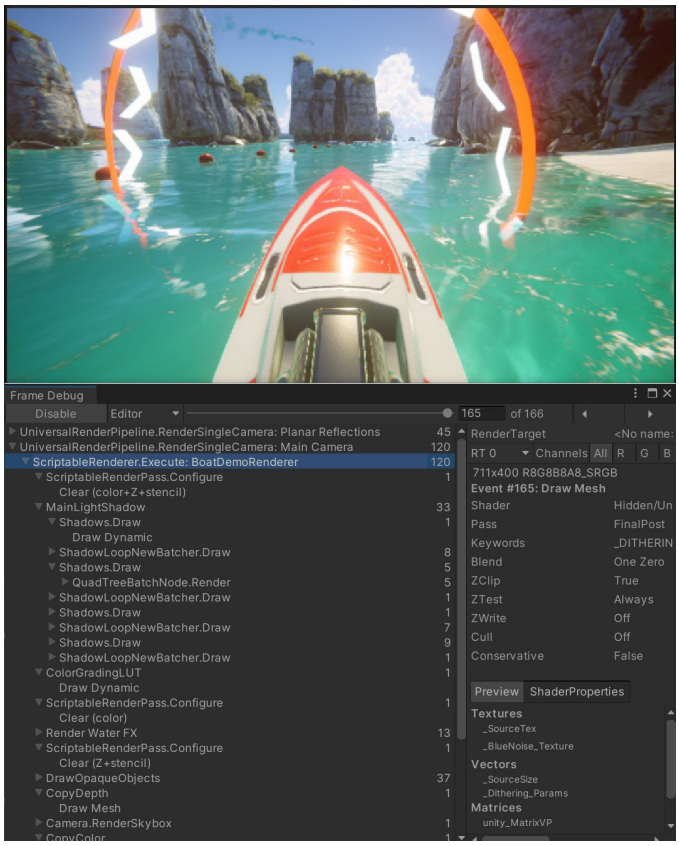


Organize your GameObjects to take advantage of these batching techniques.

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 that can help you analyze the way your game is rendered.

New to the Frame Debugger? Check out this introductory tutorial here.



The Frame Debugger breaks each frame into its separate steps.

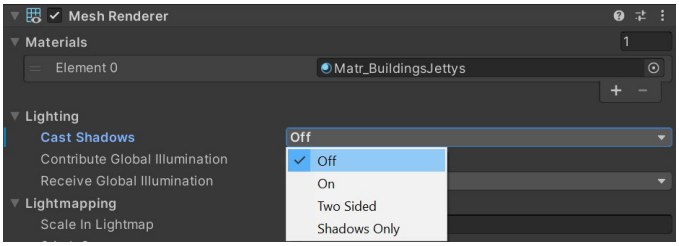
Avoid too many dynamic lights

It is crucial to avoid adding too many dynamic lights to your mobile application. 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 pipeline real-time lights.

Disable shadows

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

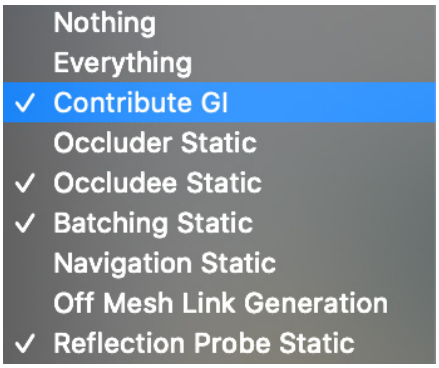


Disable shadow casting to reduce draw calls.

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.

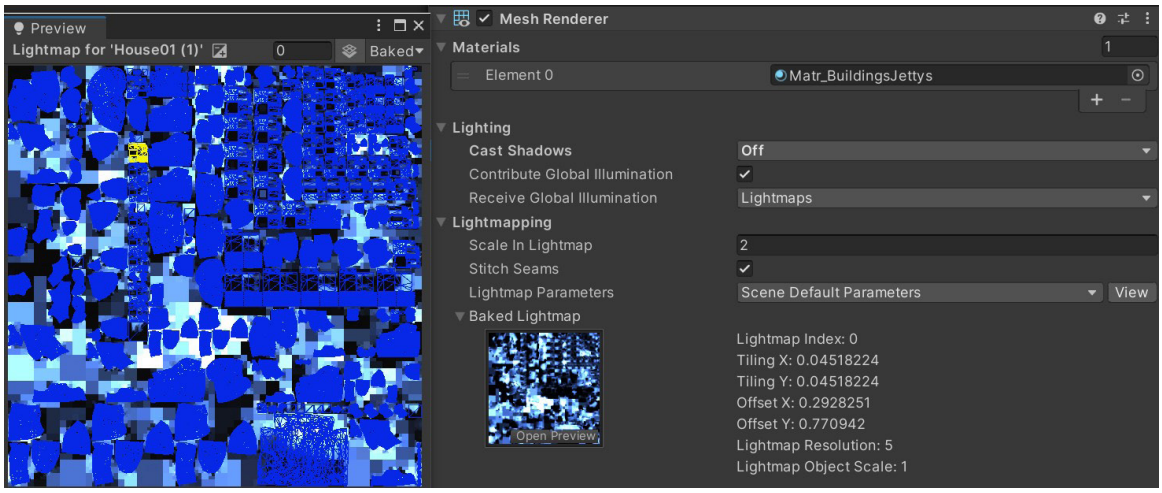
Bake your lighting into Lightmaps

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.

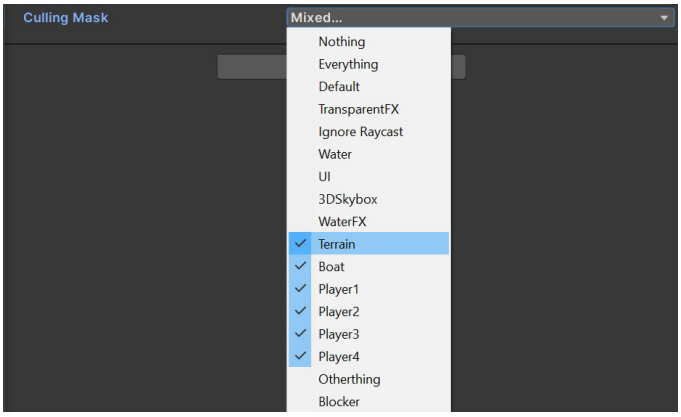


To limit memory usage, adjust the Lightmapping Settings (Windows > Rendering > Lighting Settings) and Lightmap size.

Follow the manual guide and this article on light optimization to get started with Lightmapping in Unity.

Use Light Layers

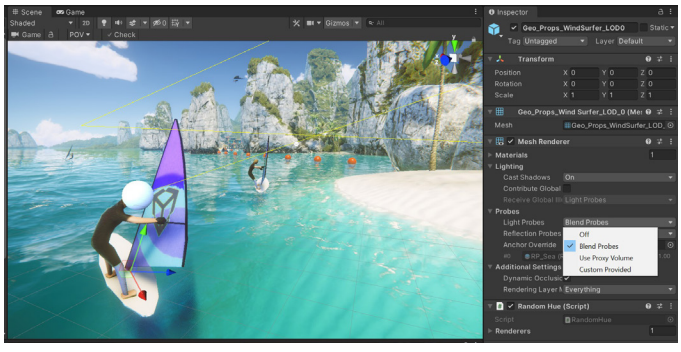
For complex scenes with multiple lights, separate your objects using layers, then confine each light's influence to a specific culling mask.



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.



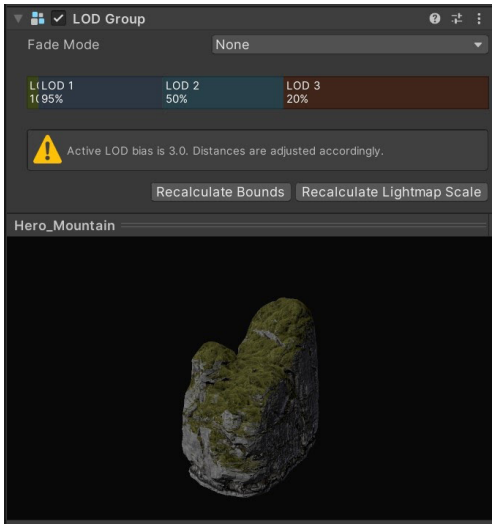
Light Probes illuminate dynamic objects in the background.

Use Level of Detail (LOD)

As objects move into the distance, Level of Detail can adjust or switch them to use simpler meshes with simpler materials and shaders, to refine GPU performance.

See the Working with LODs course on Unity Learn for more detail.

Source meshes, modeled at varying resolutions.



Example of a mesh using a Level of Detail Group.

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 through the **Window > Rendering > Occlusion Culling** dialog. Though not necessary for every scene, culling can improve performance in many cases.

Check out the Working with Occlusion Culling tutorial for more information.

Avoid mobile native resolution

With phones and tablets becoming increasingly advanced, newer devices tend to sport very high resolutions.

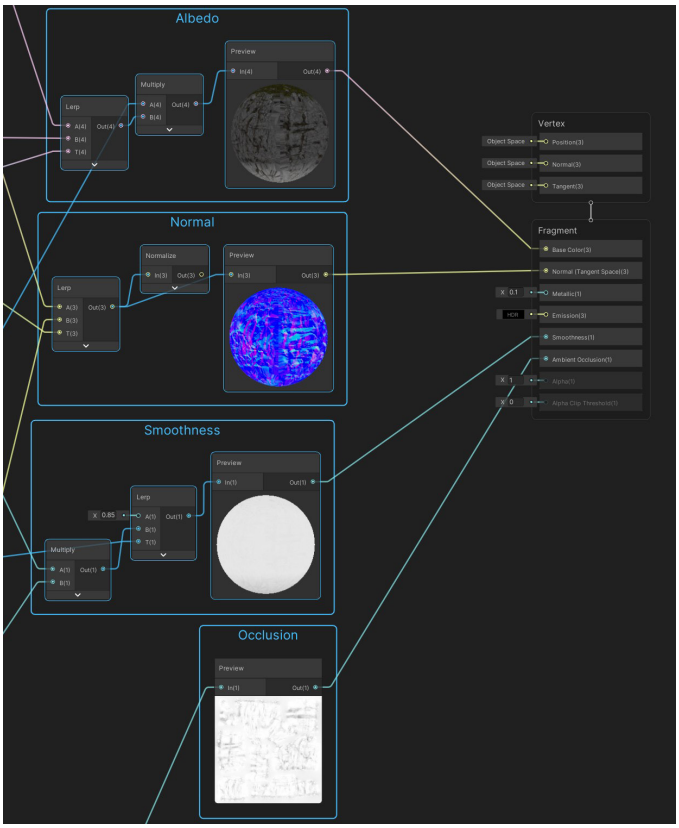
Use **Screen.SetResolution(width, height, false)** to lower the output resolution and regain some performance. Profile multiple resolutions to find the best balance between quality and speed.

Limit use of cameras

Each camera incurs some overhead, whether it's doing meaningful work or not. Only use Camera components required 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.



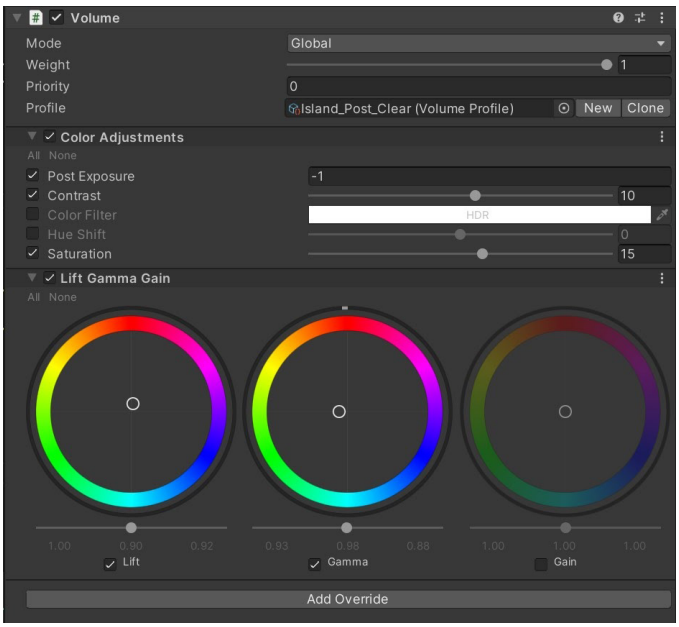
Create custom shaders with Shader Graph.

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.

Limit Post-processing effects

Fullscreen Post-processing effects, like glows, can dramatically slow down performance. Use them cautiously in your title's art direction.



Keep post-processing effects simple in mobile applications.

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, then swap to a simpler MeshRenderer at runtime.

Minimize Reflection Probes

A Reflection Probe can create realistic reflections, but can be very costly in terms of batches. Use low-resolution cubemaps, culling masks, and texture compression to improve runtime performance.

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, 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

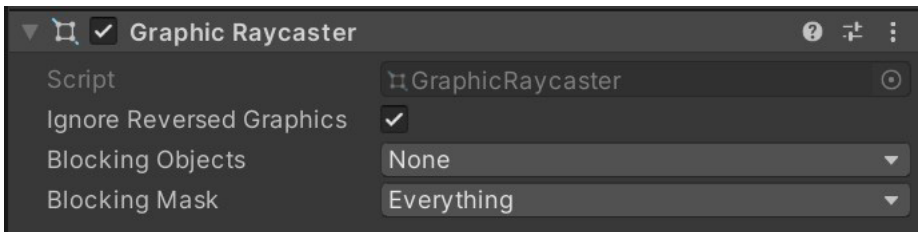
You might have UI elements that only appear sporadically in the game (e.g., a health bar that appears when a character takes damage). If your invisible UI element is active, it might still be using draw calls. Explicitly disable any invisible UI components and re-enable them as needed.

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

Limit GraphicRaycasters and disable Raycast Target

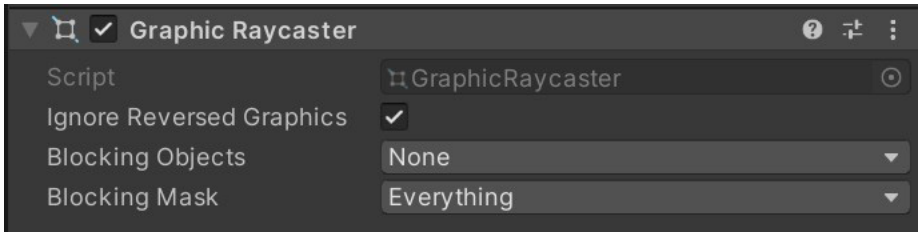
Input events like onscreen touches or clicks require the GraphicRaycaster component. This simply loops through each input point onscreen and checks if they're within a UI's RectTransform.

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 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.

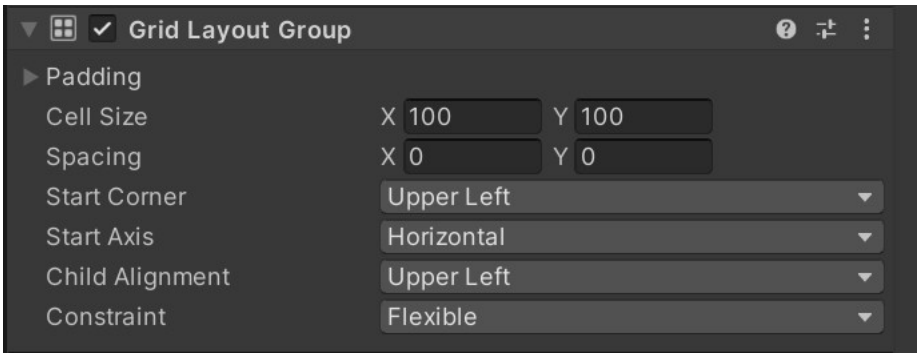


Disable Raycast Target if possible.

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. 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.

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 on each 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

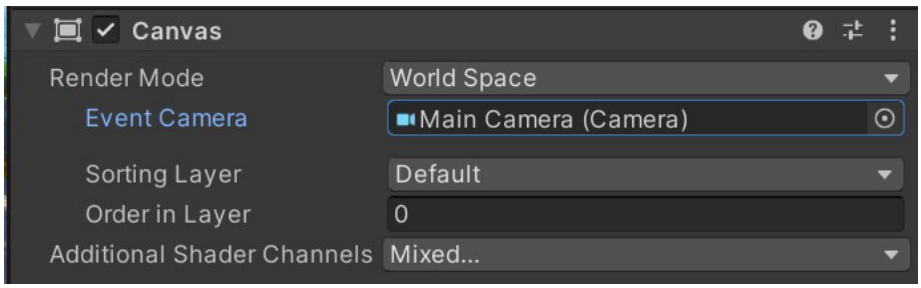
If your pause or start screen covers everything else in the scene, disable the camera that is rendering the 3D scene. Similarly, disable any background Canvas elements hidden behind the top Canvas.

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, as that does not require a camera.



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

Audio

Though audio is not typically 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. Otherwise, a multichannel sound used positionally at runtime will be flattened to a mono source, thus increasing CPU cost and wasting memory.

Use original uncompressed WAV files as your source assets

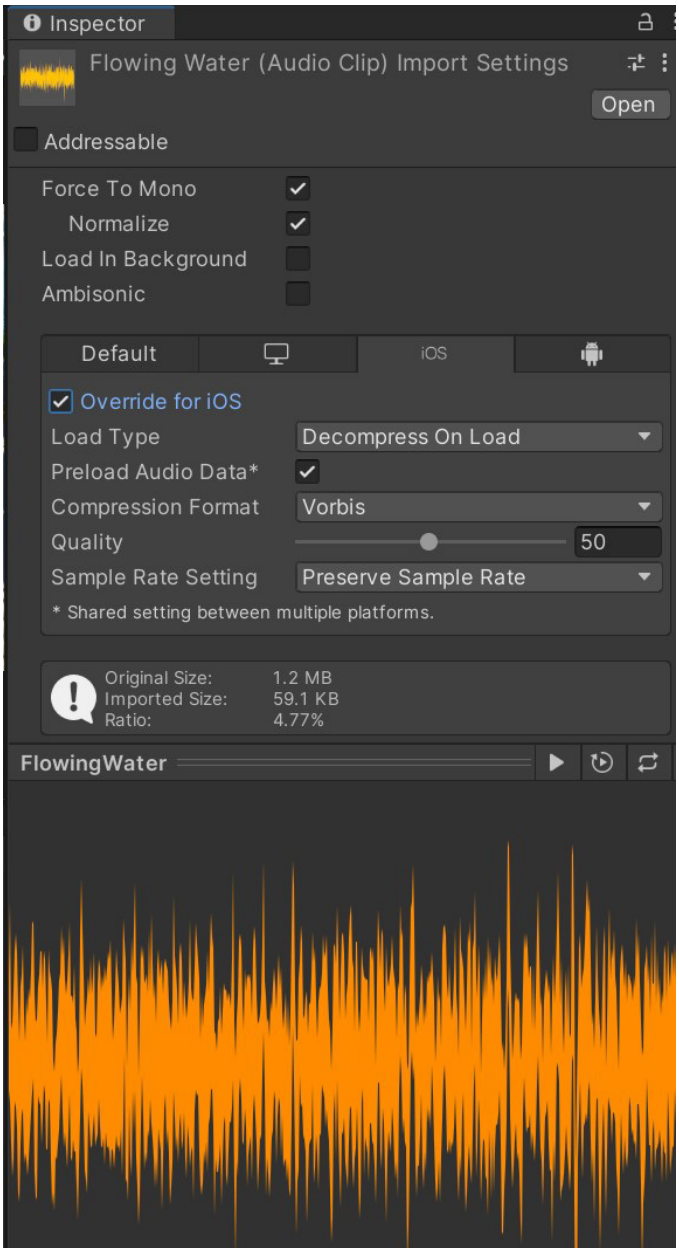
If you use any compressed format (such as MP3 or Vorbis), Unity will decompress it, then recompress it during build time. This results in two lossy passes, degrading the final quality.

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 quick to decode during playback.



Optimize the Import Settings of your AudioClips.

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 by 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**, or else 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 following settings.

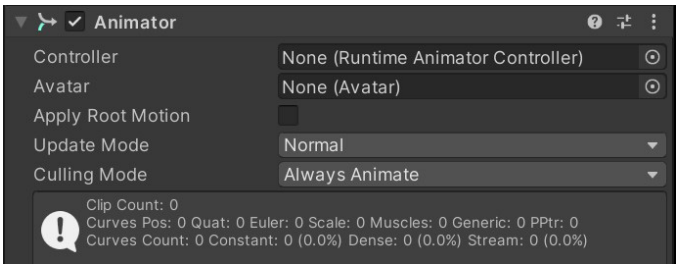
Use generic versus humanoid rigs

By default, Unity imports animated models with the Generic Rig, though 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, leverage the legacy Animation components for mobile.



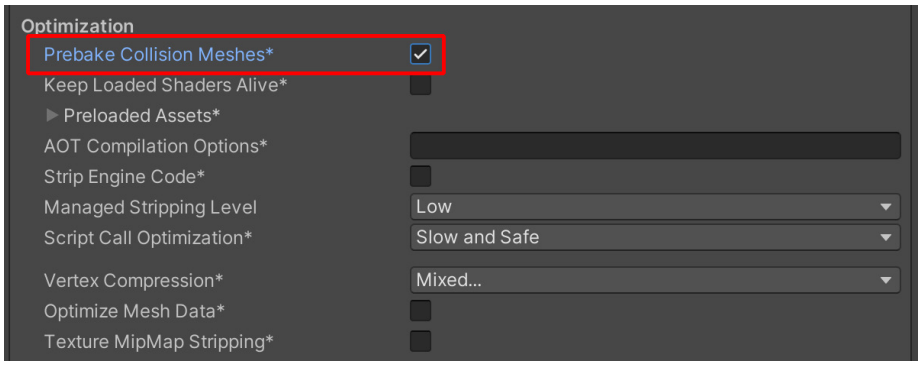
Animators are sometimes quite expensive.

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

Physics

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

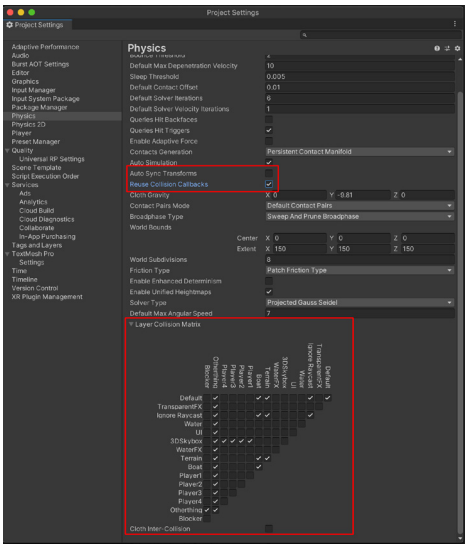
Optimize your settings



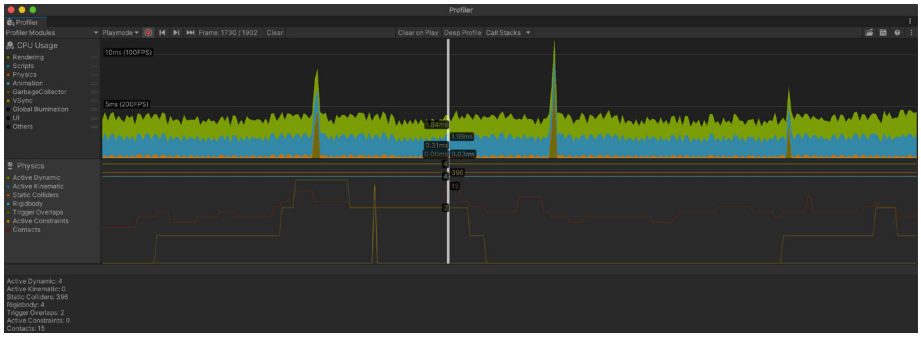
Enable Prebake Collision Meshes.

In the PlayerSettings, check Prebake Collision Meshes whenever possible.

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



Modify the physics project settings to augment performance.

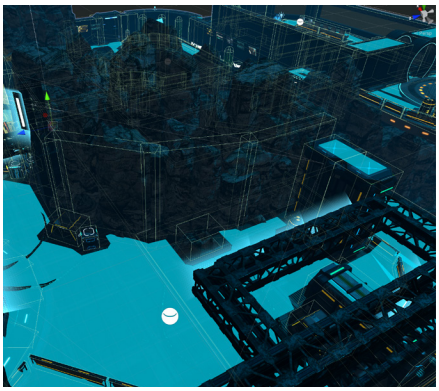


Keep an eye on the Physics module of the Profiler for performance issues.

Disable Auto Sync Transforms and enable Reuse Collision Callbacks.

Simplify colliders

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



Use primitives or simplified meshes for colliders.

Move a Rigidbody using physics methods

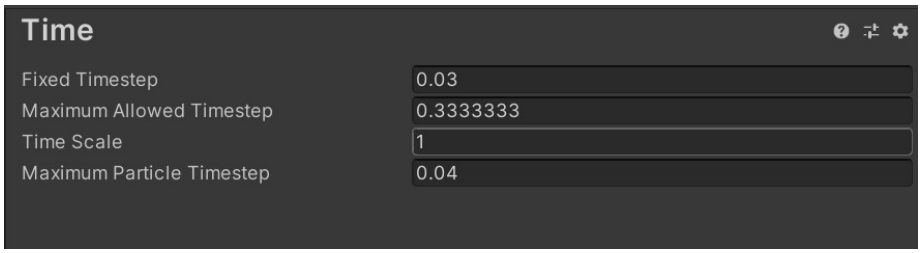
Use class methods like MovePosition or AddForce to move your Rigidbody objects. Translating their Transform components directly can lead to physics world recalculations, which are expensive in complex scenes. Move physics bodies in FixedUpdate rather than Update.

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).

If your frame rate drops at runtime, however, this means that Unity will likely call FixedUpdate multiple times per frame and potentially create a CPU performance issue with physics-heavy content.

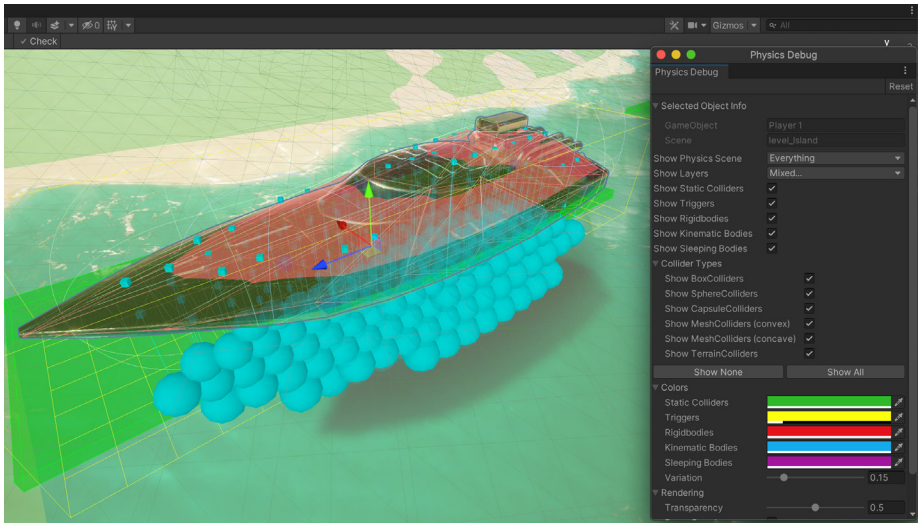
The Maximum Allowed Timestep limits how much time physics calculations and FixedUpdate events can use in the event that the frame rate drops. Lowering this value means that physics and animation can slow down, while also reducing their impact on the frame rate during a performance hitch.



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 problematic colliders or discrepancies. This shows a color-coded indicator of the GameObjects that can 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 demanding endeavor that often involves many developers. Make sure that your project is set up optimally for your team to collaborate.

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, 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 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, and ensure that you 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 that artists and designers can collaborate effectively on a single level while minimizing the risk of conflicts.

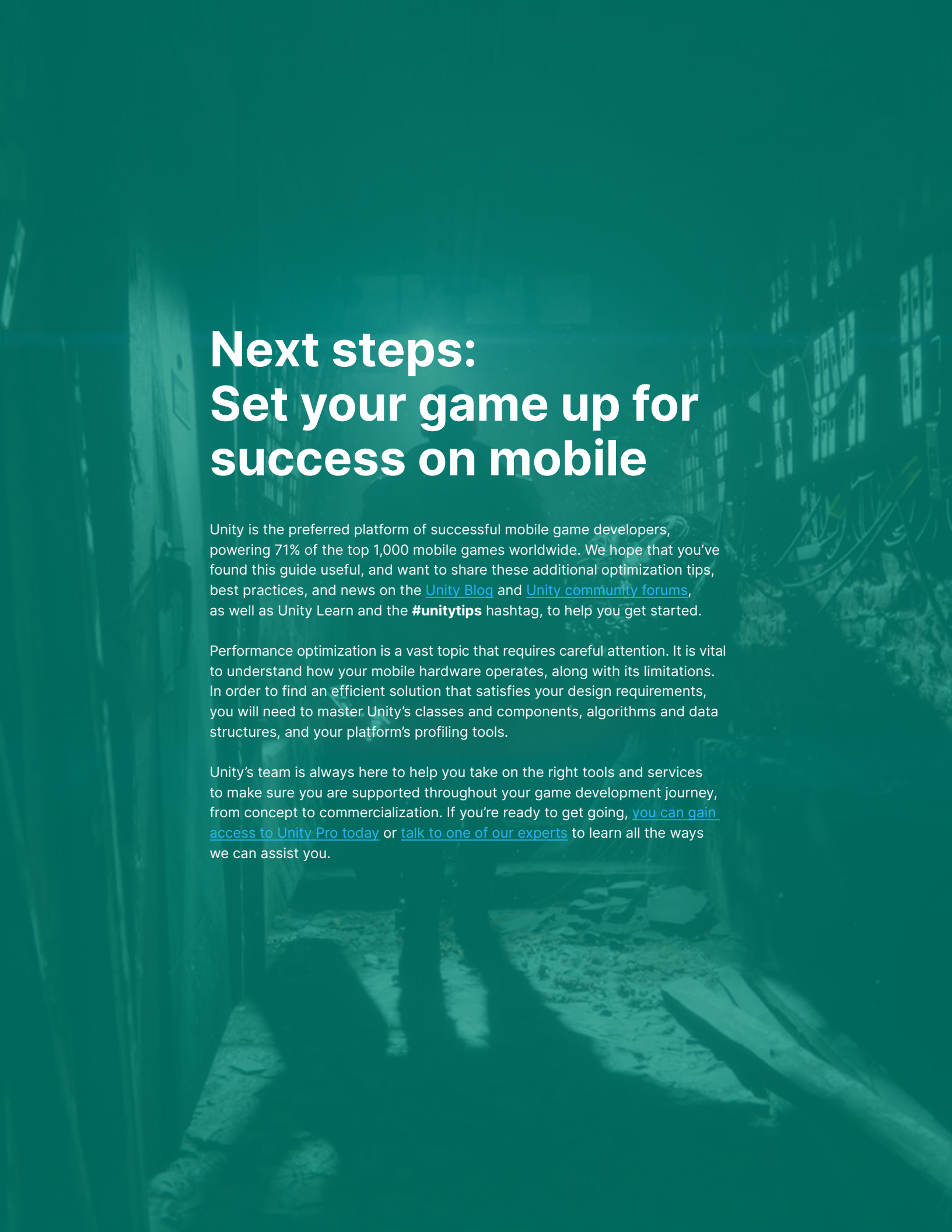
Note that, at runtime, your project can load Scenes additively using SceneManager.LoadSceneAsync passing the LoadSceneMode.Additive parameter mode.

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 art of your build if you don't remove them. Strip out any unneeded resources left over from prototyping.

Accelerate 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.



Next steps: Set your game up for success on mobile

Unity is the preferred platform of successful mobile game developers, powering 71% of the top 1,000 mobile games worldwide. We hope that you've found this guide useful, and want to share these additional optimization tips, best practices, and news on the Unity Blog and Unity community forums, as well as Unity Learn and the **#unitytips** hashtag, to help you get started.

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.

Unity's team is always here to help you take on the right tools and services to make sure you are supported throughout your game development journey, from concept to commercialization. If you're ready to get going, you can gain access to Unity Pro today or talk to one of our experts to learn all the ways we can assist you.


Unity®