UNITY FOR GAMES

UNITY 2020 LTS EDITION | E-BOOK

Optimize Your Mobile Game Performance

Contents

Introduction

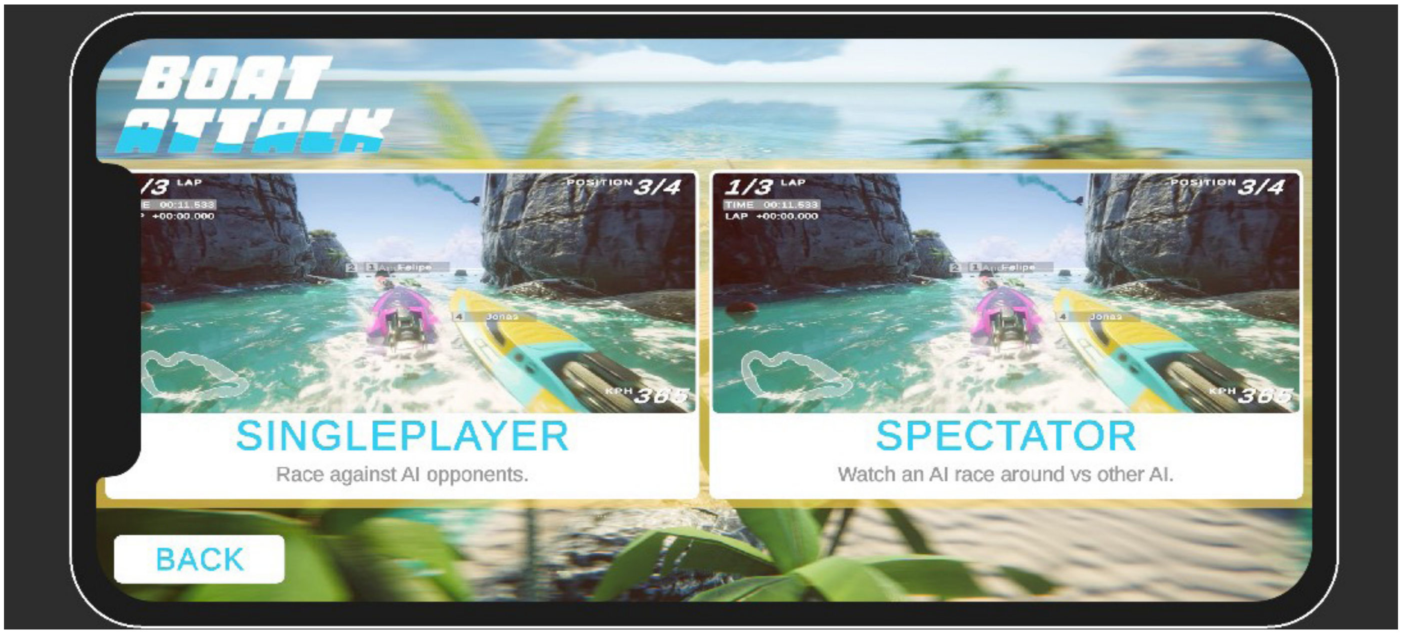
Optimizing your iOS and Android applications is an essential process that underpins the entire development cycle. Mobile hardware continues to evolve, and a mobile game's optimization – along with its art, game design, audio, and monetization strategy – plays a key role in shaping the player experience.

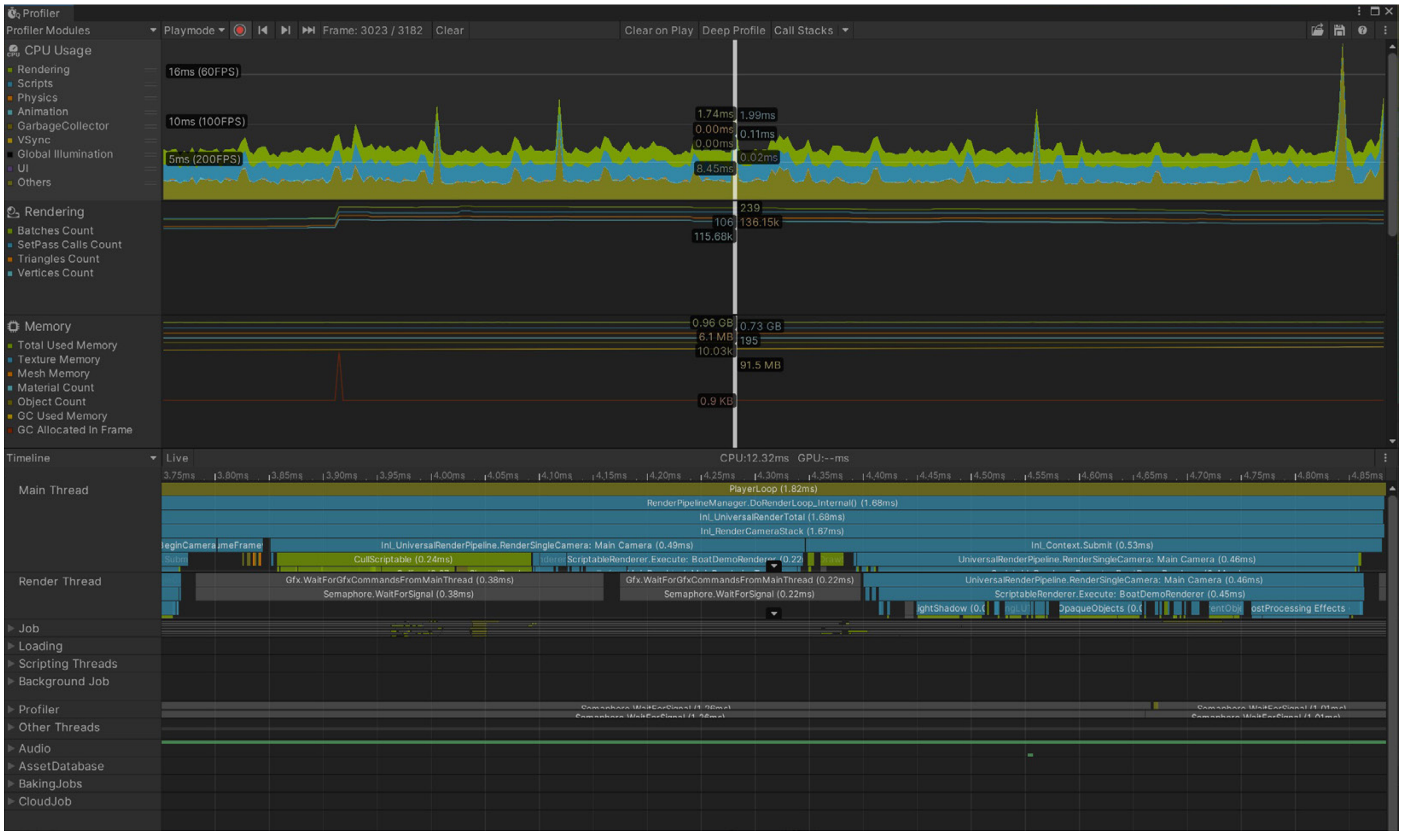
Both iOS and Android have active user bases in the *billions*. If your mobile game is highly optimized, it has a better chance at passing certification from platform-specific stores. To maximize your opportunity for success at launch and beyond, your aim is always twofold: building the slickest, most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

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.

Happy optimizing from the Unity team!


BOAT ATTACK
SINGLEPLAYER
Race against AI opponents.
SPECTATOR
Watch an AI race around vs other AI.
BACK



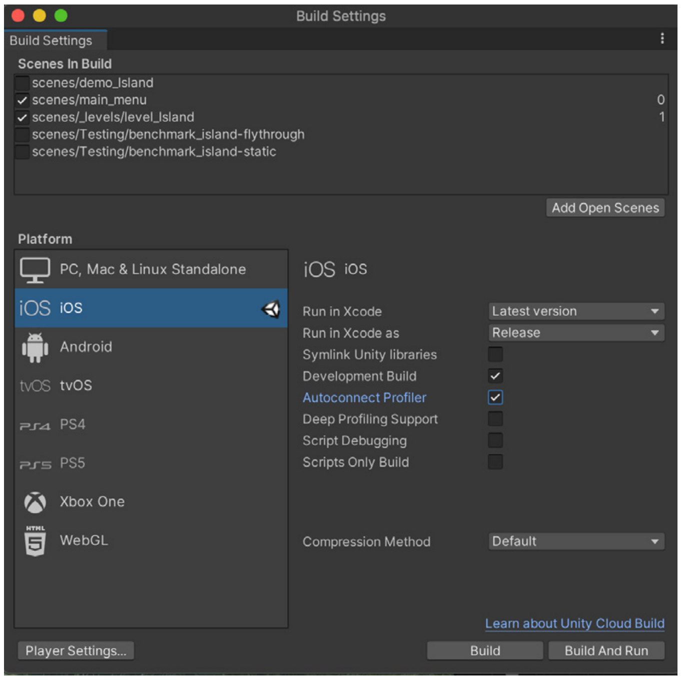
Use the Unity Profiler to test performance and resource allocation for your application.

Profiling

The Unity Profiler can help you detect the causes of any lags or freezes during runtime or understand what's happening at a specific frame (point in time). Enable the CPU and Memory tracks by default. You can monitor additional Profiler Modules (such as Renderer, Audio, Physics, etc.) if you have specific needs for your game (e.g., physics-heavy or music-based gameplay).

Build the application to your device by checking **Development Build** and **Autoconnect Profiler**, or connect manually to accelerate app startup time.

Choose the 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. This means that the Unity Editor has to do more CPU work and takes up more memory, but it can be useful depending on your specific scenario.



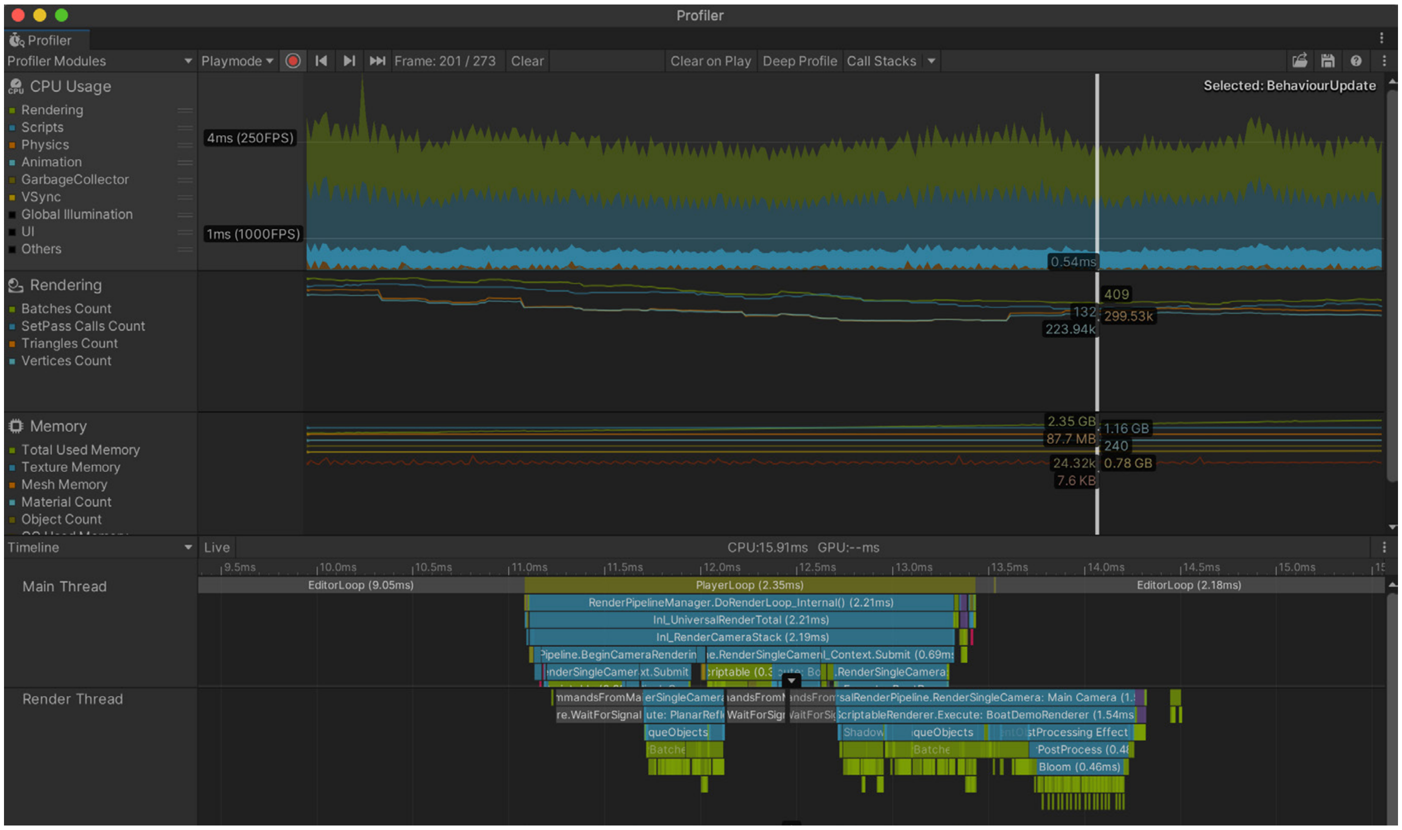
Build Settings

Scenes In Build

- [ ] scenes/demo_Island
- [✓] scenes/main_menu 0
- [✓] scenes/_levels/level_Island 1
- [ ] scenes/Testing/benchmark_island-flythrough
- [ ] scenes/Testing/benchmark_island-static

Platform: iOS

Run in Xcode	Latest version
Run in Xcode as	Release
Symlink Unity libraries	☐
Development Build	✓
Autoconnect Profiler	✓
Deep Profiling Support	☐
Script Debugging	☐
Scripts Only Build	☐
Compression Method	Default



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

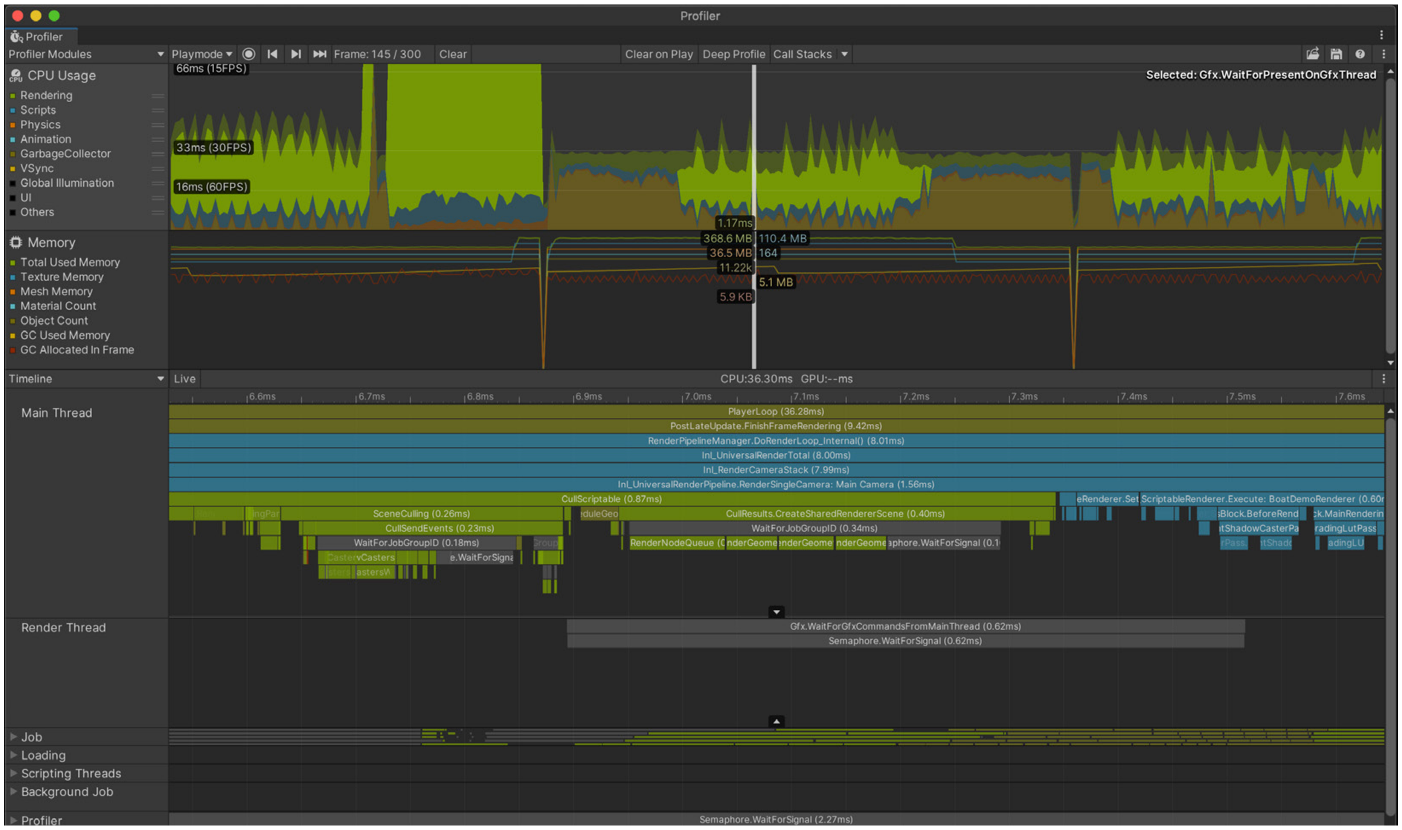
This is an instrumentation-based profiler that profiles code timings explicitly wrapped in ProfileMarkers (such as Monobehaviour's Start or Update methods or specific API calls). Also, when you're using the Deep Profiling setting, Unity can profile the beginning and end of every function call in your script code to tell you exactly which part of your application is causing a slowdown (but this comes with extra overhead).

When profiling your game, we recommend that you cover both spikes and the cost of an average frame in your game. Understanding and optimizing expensive operations that occur each frame can be more useful for applications running below the target framerate. When looking for spikes, explore expensive operations first (e.g., physics, AI, animation) and garbage collection.

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

— **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 this frame.

— **Timeline** shows a visual breakdown of the specific frame's timings. This allows you to visualize how the activities relate to one another and across different threads. Use this to determine if you are CPU-bound or GPU-bound.



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.

Profile early and often

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" of your project, you'll be able to spot new issues more easily.

Don't optimize blindly

Don't guess or make assumptions about what is slowing your game's performance. Use the Unity Profiler and platform-specific tools to locate what, specifically, is causing the lag.

Also, 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 those.

Profile on the target device

While profiling in the Editor can give you a very rough idea of the relative performance of different systems in your game, there's no substitute for the real thing. Profile a development build on target devices whenever possible. Remember to profile and optimize for the lowest-spec device you plan to support.

The Unity Profiler alone cannot see into every part of the engine. Fortunately, iOS and Android both include native tools to help you test performance:

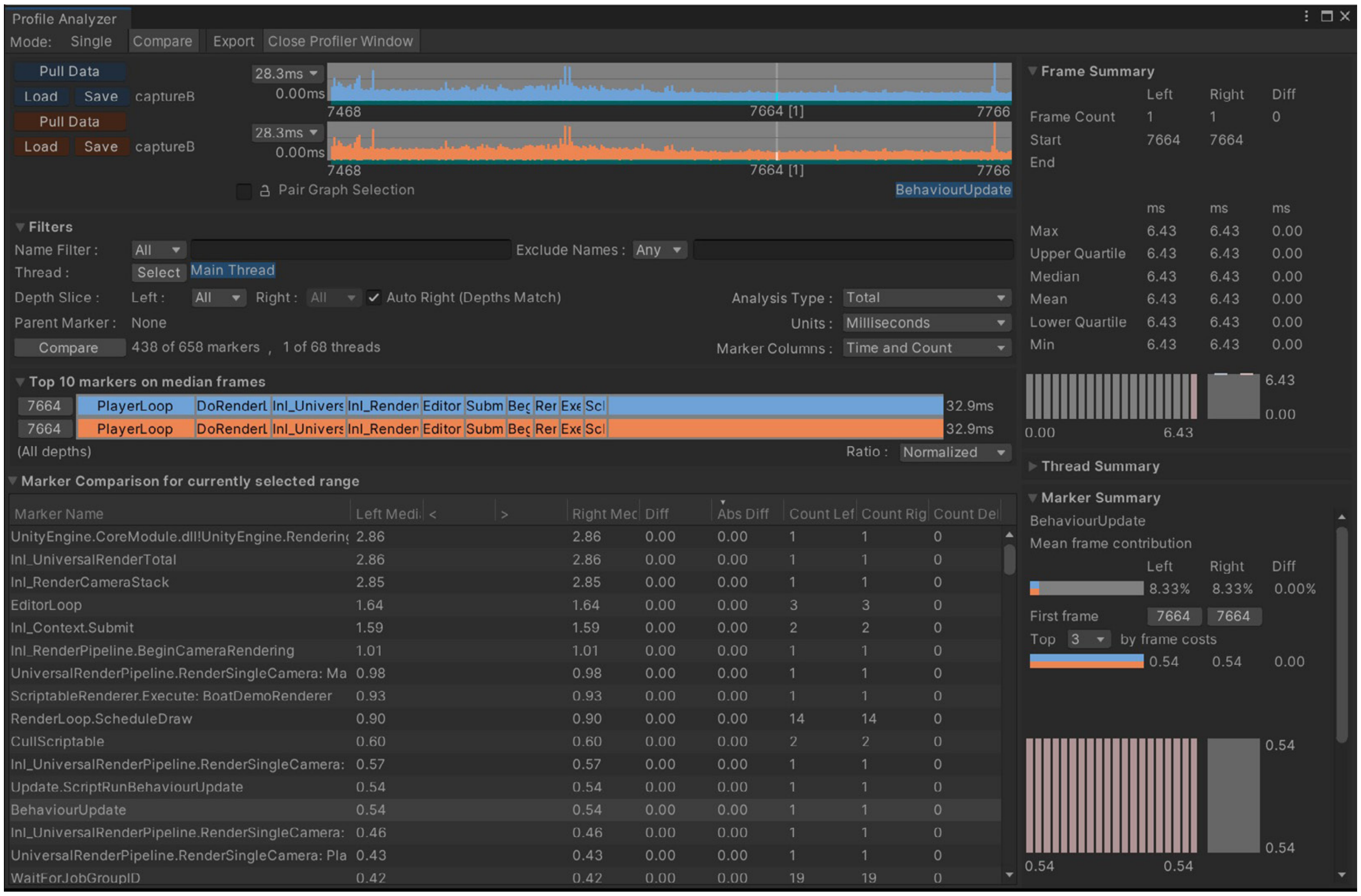
— On iOS, use Xcode and Instruments.

— On Android, use Android Studio and Android Profiler.

Certain hardware can also 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.

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 diff two data sets, which is vital for testing changes and showing improvements. The Profile Analyzer is available via Unity's Package Manager.



Filters
Name Filter : All Exclude Names : Any
Thread : Select Main Thread
Depth Slice : Left : All Right : All ✓ Auto Right (Depths Match)
Parent Marker : None
Analysis Type : Total
Units : Milliseconds
Marker Columns : Time and Count
Compare 438 of 658 markers , 1 of 68 threads

Top 10 markers on median frames

7664	PlayerLoop DoRenderL Inl_Univers Inl_Render Editor Subm Beg Ren Exe Scl	32.9ms
7664	PlayerLoop DoRenderL Inl_Univers Inl_Render Editor Subm Beg Ren Exe Scl	32.9ms

(All depths) Ratio : Normalized

Marker Comparison for currently selected range

Marker Name	Left Medi.	<	>	Right Med	Diff	Abs Diff	Count Lef	Count Rig	Count Del
UnityEngine.CoreModule.dll!UnityEngine.Rendering	2.86			2.86	0.00	0.00	1	1	0
Inl_UniversalRenderTotal	2.86			2.86	0.00	0.00	1	1	0
Inl_RenderCameraStack	2.85			2.85	0.00	0.00	1	1	0
EditorLoop	1.64			1.64	0.00	0.00	3	3	0
Inl_Context.Submit	1.59			1.59	0.00	0.00	2	2	0
Inl_RenderPipeline.BeginCameraRendering	1.01			1.01	0.00	0.00	1	1	0
UniversalRenderPipeline.RenderSingleCamera: Ma	0.98			0.98	0.00	0.00	1	1	0
ScriptableRenderer.Execute: BoatDemoRenderer	0.93			0.93	0.00	0.00	1	1	0
RenderLoop.ScheduleDraw	0.90			0.90	0.00	0.00	14	14	0
CullScriptable	0.60			0.60	0.00	0.00	2	2	0
Inl_UniversalRenderPipeline.RenderSingleCamera:	0.57			0.57	0.00	0.00	1	1	0
Update.ScriptRunBehaviourUpdate	0.54			0.54	0.00	0.00	1	1	0
BehaviourUpdate	0.54			0.54	0.00	0.00	1	1	0
Inl_UniversalRenderPipeline.RenderSingleCamera:	0.46			0.46	0.00	0.00	1	1	0
UniversalRenderPipeline.RenderSingleCamera: Pla	0.43			0.43	0.00	0.00	1	1	0
WaitForJobGroupID	0.42			0.42	0.00	0.00	19	19	0

Frame Summary

	Left	Right	Diff
Frame Count	1	1	0
Start	7664	7664	
End			
	ms	ms	ms
Max	6.43	6.43	0.00
Upper Quartile	6.43	6.43	0.00
Median	6.43	6.43	0.00
Mean	6.43	6.43	0.00
Lower Quartile	6.43	6.43	0.00
Min	6.43	6.43	0.00

Thread Summary

Marker Summary
BehaviourUpdate
Mean frame contribution

	Left	Right	Diff
	8.33%	8.33%	0.00%
First frame	7664	7664	
Top 3 by frame costs	0.54	0.54	0.00

Need a deeper dive into frames and marker data? The Profile Analyzer 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.

For mobile, however, you cannot use this time consistently because the device will overheat and the OS will thermal throttle the CPU and GPU. We recommend that you only use around 65% of the available time to allow 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 for short periods of time (e.g., for cutscenes or loading sequences) but not for a prolonged duration.

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.

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.

Account for device temperature

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 may report poor performance, even if it may not be cause for concern. Combat profiling overhead by profiling in short bursts to keep the device cool and simulate real-world conditions.

Test on a min-spec device

There is a wide range of iOS and Android devices. Test your project on the minimum device specifications that you want your application to support.

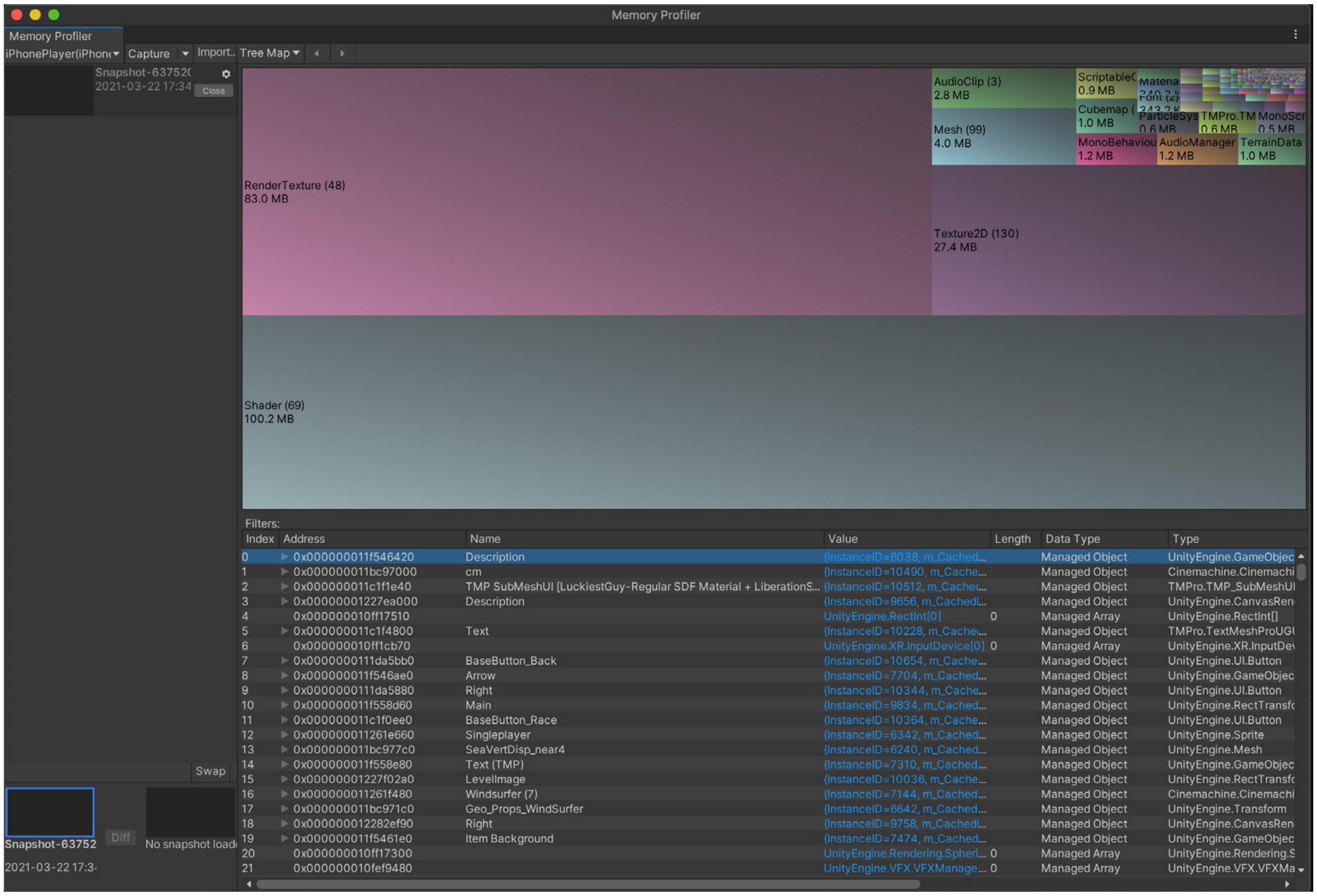
Memory

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 (GC) 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 minimizing the effect of garbage collection.

See Understanding the managed heap for more information.



Filters:

Index	Address	Name	Value	Length	Data Type	Type
0	0x000000011f546420	Description	{InstanceID=8038, m_Cached...		Managed Object	UnityEngine.GameObjec
1	0x000000011bc97000	cm	{InstanceID=10490, m_Cache...		Managed Object	Cinemachine.Cinemachi
2	0x000000011c1f1e40	TMP SubMeshUI [LuckiestGuy-Regular SDF Material + LiberationS...	{InstanceID=10512, m_Cache...		Managed Object	TMPro.TMP_SubMeshUI
3	0x00000001227ea000	Description	{InstanceID=9656, m_CachedI...		Managed Object	UnityEngine.CanvasRen
4	0x000000010ff17510		UnityEngine.RectInt[0]	0	Managed Array	UnityEngine.RectInt[]
5	0x000000011c1f4800	Text	{InstanceID=10228, m_Cache...		Managed Object	TMPro.TextMeshProUGU
6	0x000000010ff1cb70		UnityEngine.XR.InputDevice[0]	0	Managed Array	UnityEngine.XR.InputDev
7	0x0000000111da5bb0	BaseButton_Back	{InstanceID=10654, m_Cache...		Managed Object	UnityEngine.UI.Button
8	0x000000011f546ae0	Arrow	{InstanceID=7704, m_Cached...		Managed Object	UnityEngine.GameObjec
9	0x0000000111da5880	Right	{InstanceID=10344, m_Cache...		Managed Object	UnityEngine.UI.Button
10	0x000000011f558d60	Main	{InstanceID=9834, m_Cached...		Managed Object	UnityEngine.RectTransfc
11	0x000000011c1f0ee0	BaseButton_Race	{InstanceID=10364, m_Cache...		Managed Object	UnityEngine.UI.Button
12	0x000000011261e660	Singleplayer	{InstanceID=6342, m_Cached...		Managed Object	UnityEngine.Sprite
13	0x000000011bc977c0	SeaVertDisp_near4	{InstanceID=6240, m_Cached...		Managed Object	UnityEngine.Mesh
14	0x000000011f558e80	Text (TMP)	{InstanceID=7310, m_Cached...		Managed Object	UnityEngine.GameObjec
15	0x00000001227f02a0	LevelImage	{InstanceID=10036, m_Cache...		Managed Object	UnityEngine.RectTransfc
16	0x000000011261f480	Windsurfer (7)	{InstanceID=7144, m_Cached...		Managed Object	Cinemachine.Cinemachi
17	0x000000011bc971c0	Geo_Props_WindSurfer	{InstanceID=6642, m_Cached...		Managed Object	UnityEngine.Transform
18	0x000000012282ef90	Right	{InstanceID=9758, m_CachedI...		Managed Object	UnityEngine.CanvasRen
19	0x000000011f5461e0	Item Background	{InstanceID=7474, m_Cached...		Managed Object	UnityEngine.GameObjec
20	0x000000010ff17300		UnityEngine.Rendering.Spheri...	0	Managed Array	UnityEngine.Rendering.S
21	0x000000010fef9480		UnityEngine.VFX.VFXManage...	0	Managed Array	UnityEngine.VFX.VFXMa

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, helping you spot problems like fragmentation and memory leaks.

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

Watch how you can use the Memory Profiler in Unity to improve memory usage. You can also check out the 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 when it has finished its work.

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.
- **Coroutines:** 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.

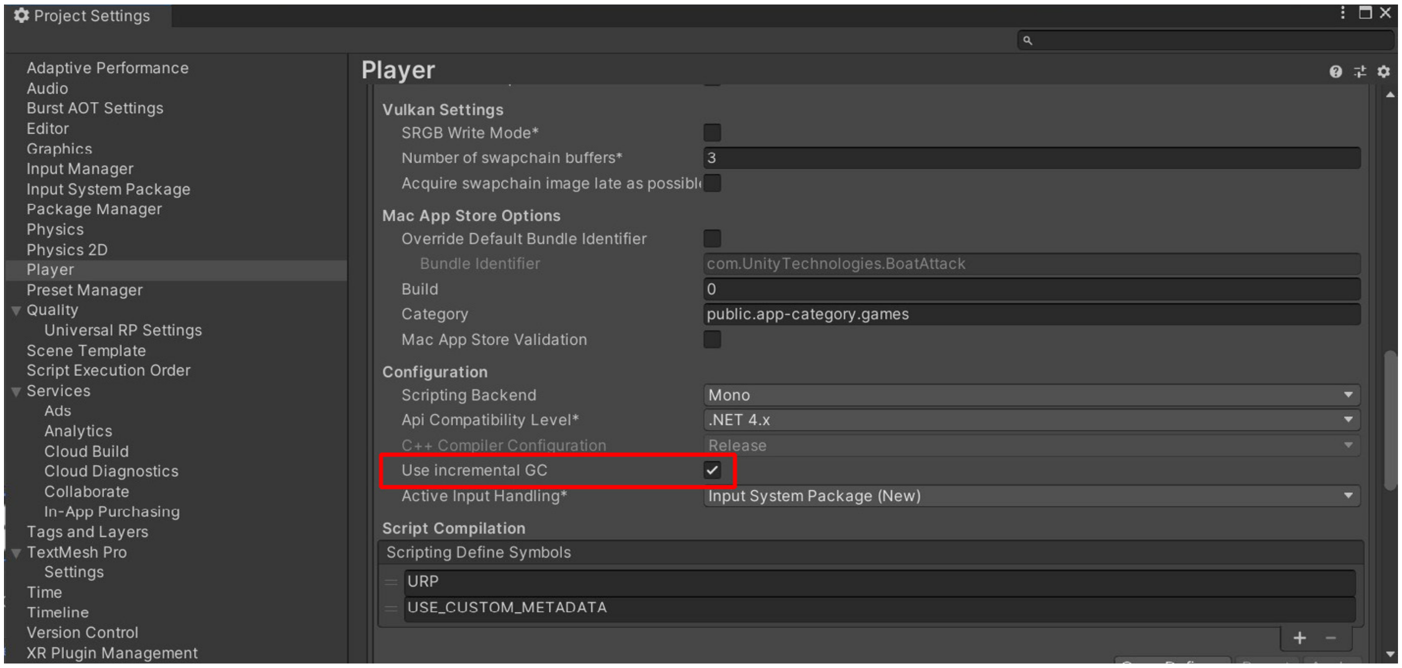
Time garbage collection if possible

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 where you could potentially use this to your advantage.

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.



Player

Vulkan Settings
- SRGB Write Mode*: ☐
- Number of swapchain buffers*: 3
- Acquire swapchain image late as possibl: ☐

Mac App Store Options
- Override Default Bundle Identifier: ☐
- Bundle Identifier: com.UnityTechnologies.BoatAttack
- Build: 0
- Category: public.app-category.games
- Mac App Store Validation: ☐

Configuration
- Scripting Backend: Mono
- Api Compatibility Level*: .NET 4.x
- C++ Compiler Configuration: Release
- Use incremental GC: ✓
- Active Input Handling*: Input System Package (New)

Script Compilation
- Scripting Define Symbols
  - URP
  - USE_CUSTOM_METADATA

Use the incremental garbage collector to reduce GC spikes.

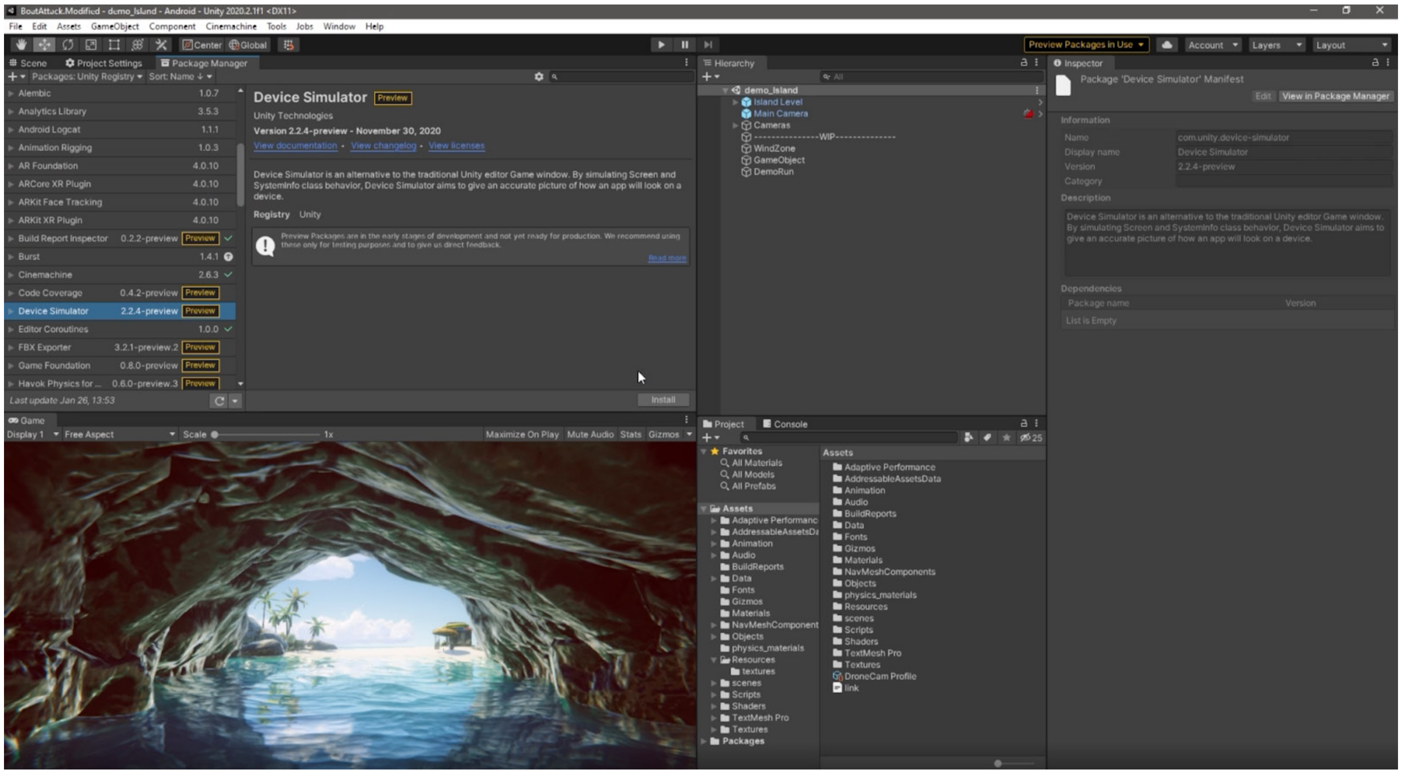
Adaptive Performance

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.



Alembic	1.0.7	
Analytics Library	3.5.3	
Android Logcat	1.1.1	
Animation Rigging	1.0.3	
AR Foundation	4.0.10	
ARCore XR Plugin	4.0.10	
ARKit Face Tracking	4.0.10	
ARKit XR Plugin	4.0.10	
Build Report Inspector	0.2.2-preview	Preview ✓
Burst	1.4.1	
Cinemachine	2.6.3	✓
Code Coverage	0.4.2-preview	Preview
Device Simulator	2.2.4-preview	Preview
Editor Coroutines	1.0.0	✓
FBX Exporter	3.2.1-preview.2	Preview
Game Foundation	0.8.0-preview	Preview
Havok Physics for ...	0.6.0-preview.3	Preview

Last update Jan 26, 13:53

Device Simulator Preview

Unity Technologies

Version 2.2.4-preview - November 30, 2020

View documentation · View changelog · View licenses

Device Simulator is an alternative to the traditional Unity editor Game window. By simulating Screen and SystemInfo class behavior, Device Simulator aims to give an accurate picture of how an app will look on a device.

Registry Unity

Preview Packages are in the early stages of development and not yet ready for production. We recommend using these only for testing purposes and to give us direct feedback.

Read more

Package 'Device Simulator' Manifest

Edit | View in Package Manager

Information

Name	com.unity.device-simulator
Display name	Device Simulator
Version	2.2.4-preview
Category	

Description

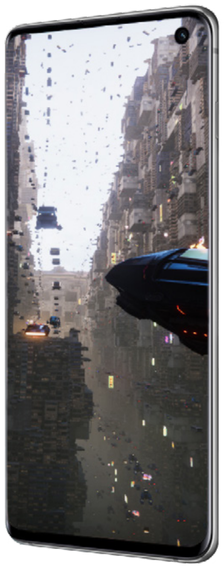
Device Simulator is an alternative to the traditional Unity editor Game window. By simulating Screen and SystemInfo class behavior, Device Simulator aims to give an accurate picture of how an app will look on a device.

Dependencies

Package name	Version
List is Empty	

Note that Adaptive Performance only works for Samsung devices.

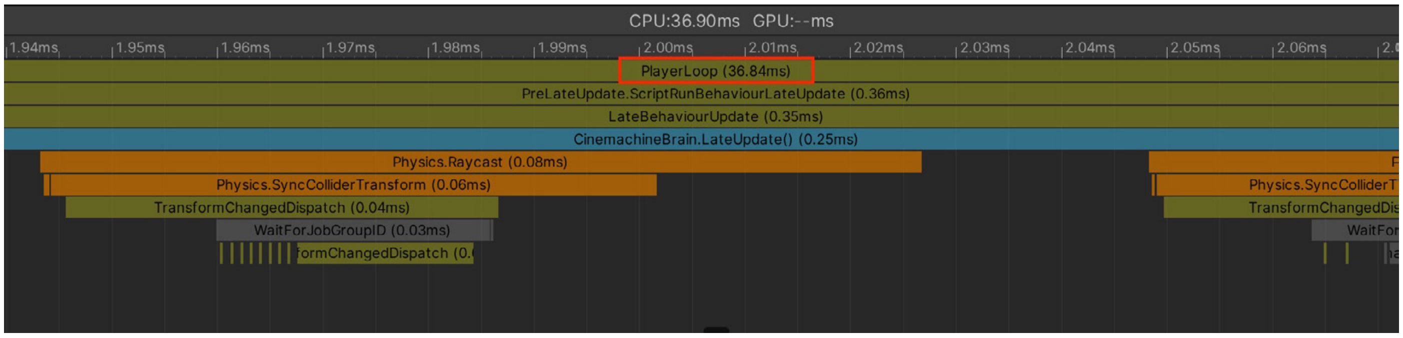
To learn more about Adaptive Performance, you can view the samples we've provided in the Package Manager by selecting **Package Manager > Adaptive Performance > Samples**. Each sample interacts with a specific scaler, so you can see how the different scalers impact your game. We also recommend reviewing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



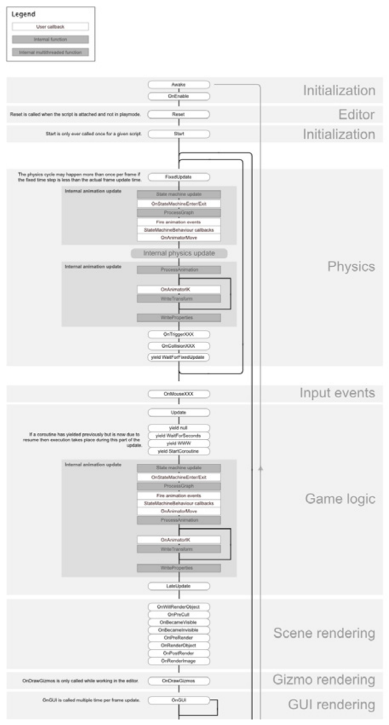
Programming and code architecture

The Unity PlayerLoop contains functions for interacting with the core of the game engine. This 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).


CPU:36.90ms GPU:--ms
1.94ms
1.95ms
1.96ms
1.97ms
1.98ms
1.99ms
2.00ms
2.01ms
2.02ms
2.03ms
2.04ms
2.05ms
2.06ms
PlayerLoop (36.84ms)
PreLateUpdate.ScriptRunBehaviourLateUpdate (0.36ms)
LateBehaviourUpdate (0.35ms)
CinemachineBrain.LateUpdate() (0.25ms)
Physics.Raycast (0.08ms)
Physics.SyncColliderTransform (0.06ms)
TransformChangedDispatch (0.04ms)
WaitForJobGroupID (0.03ms)

Your custom scripts, settings, and graphics can significantly impact how long each frame takes to calculate and render onscreen.



Understand 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.

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();
    }
}
```

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
```

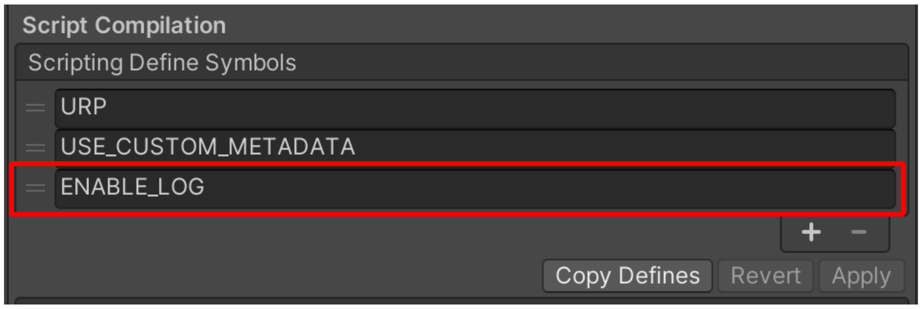
Here, you can freely use the **Update** in the Editor for testing without unnecessary overhead slipping into your build.

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);
    }
}
```


Script Compilation

Scripting Define Symbols

URP

USE_CUSTOM_METADATA

ENABLE_LOG

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.

Whenever using a Set or Get method on an Animator, Material, or Shader, use 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 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);
}
```

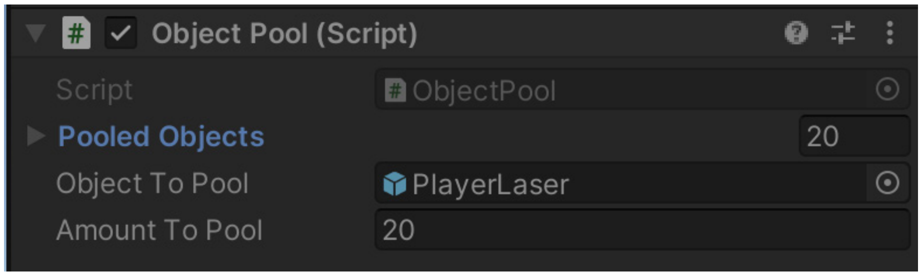
Instead, you can invoke **GetComponent** only once, as the result of the function is cached. The cached result can be reused in **Update** without any further calls to **GetComponent**.

```
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. Instead of instantiating and destroying GameObjects frequently (e.g., shooting bullets from a gun), use pools of preallocated objects which can be reused and recycled.


Object Pool (Script)

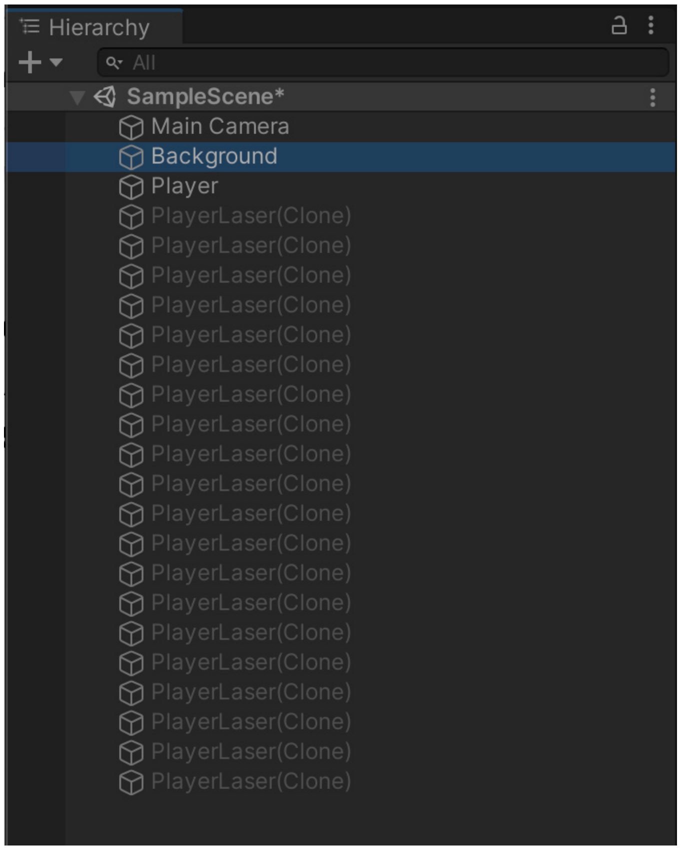
Script	ObjectPool
Pooled Objects	20
Object To Pool	PlayerLaser
Amount To Pool	20

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, simply enable the next available instance when needed, disable objects instead of destroying them, and return them to the pool.

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

Learn how to create a simple Object Pooling system in Unity here.

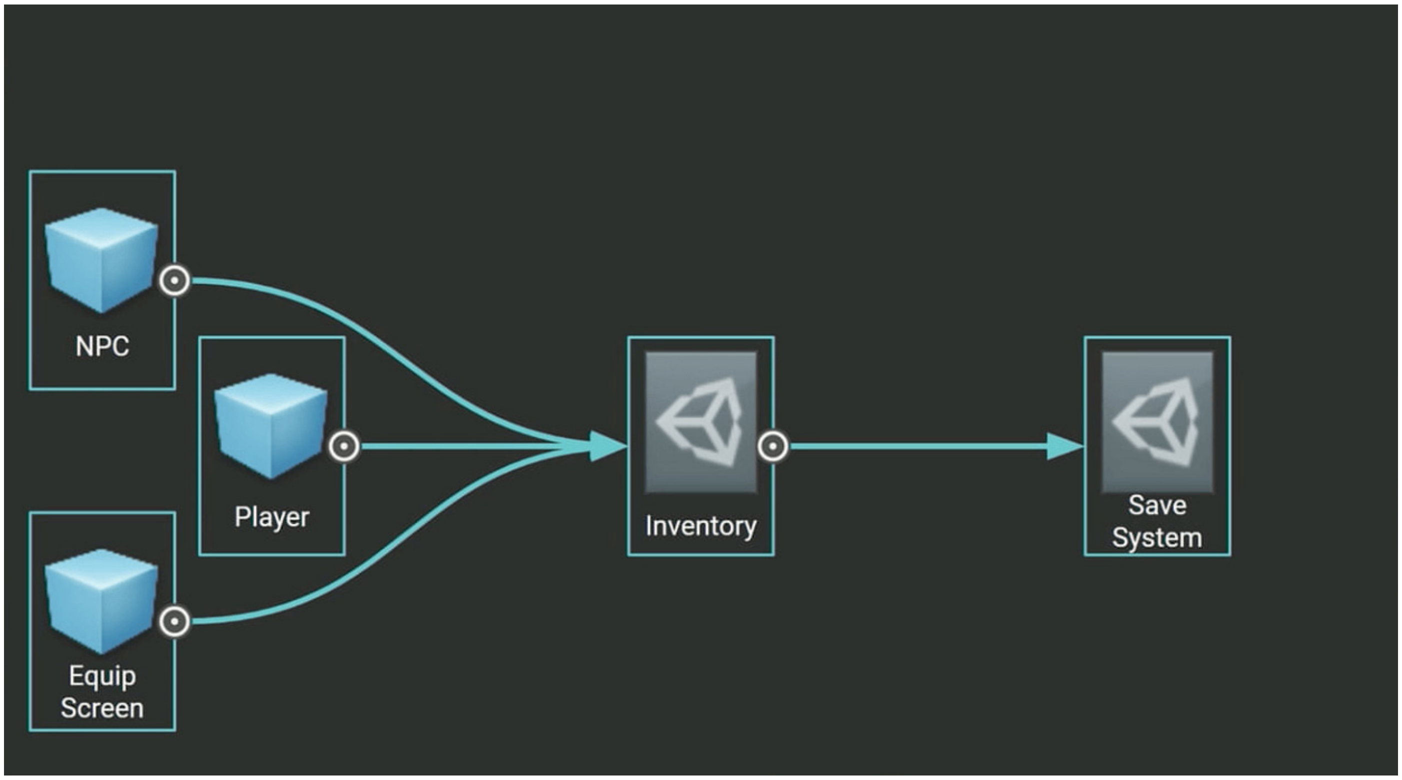


The pool of PlayerLaser objects is inactive and ready to shoot.

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.

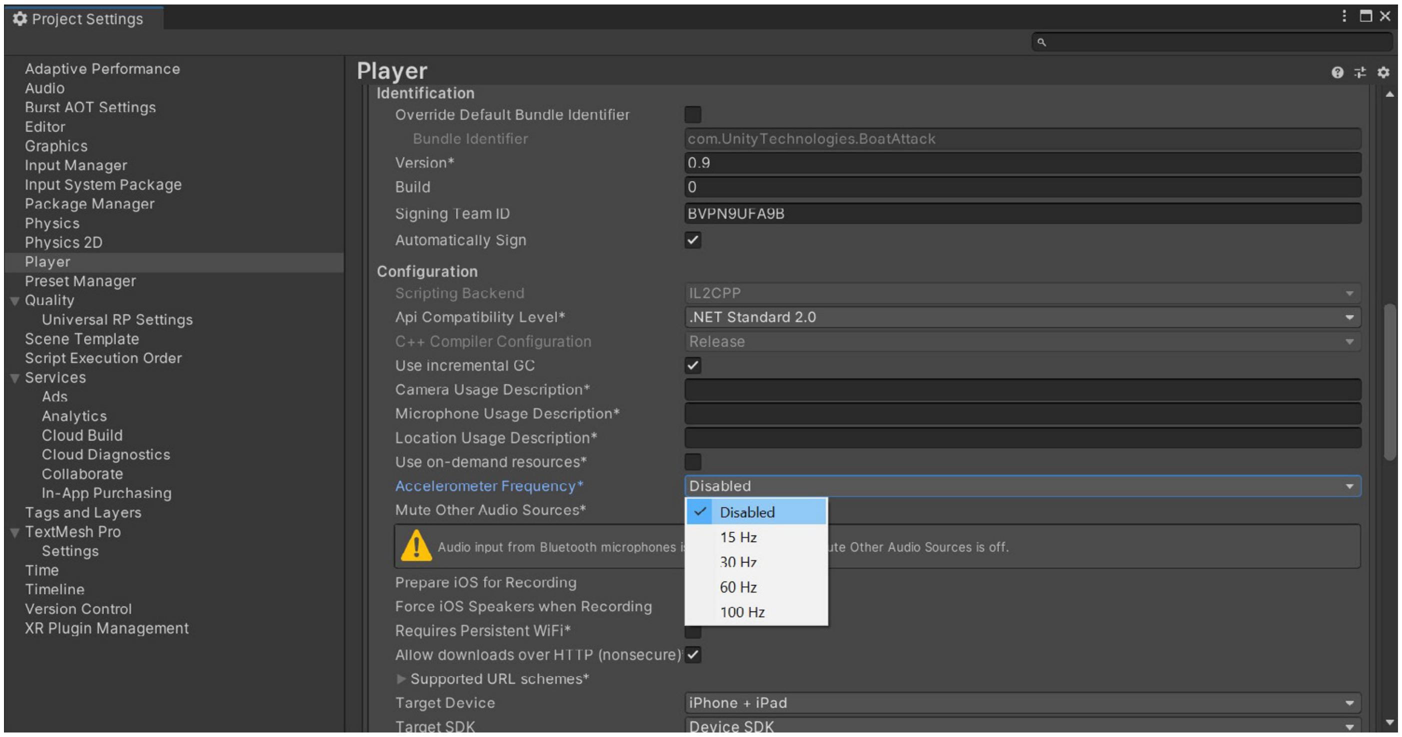
Watch this Introduction to ScriptableObjects tutorial to see how ScriptableObjects can help your project. You can also find documentation here.

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.



Override Default Bundle Identifier
Bundle Identifier: com.UnityTechnologies.BoatAttack
Version*: 0.9
Build: 0
Signing Team ID: BVPN9UFA9B
Automatically Sign: ✓
Scripting Backend: IL2CPP
Api Compatibility Level*: .NET Standard 2.0
C++ Compiler Configuration: Release
Use incremental GC: ✓
Camera Usage Description*
Microphone Usage Description*
Location Usage Description*
Use on-demand resources*
Accelerometer Frequency*: Disabled
Mute Other Audio Sources*
Audio input from Bluetooth microphones i… ute Other Audio Sources is off.
Prepare iOS for Recording
Force iOS Speakers when Recording
Requires Persistent WiFi*
Allow downloads over HTTP (nonsecure): ✓
Supported URL schemes*
Target Device: iPhone + iPad
Target SDK: Device SDK

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 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.

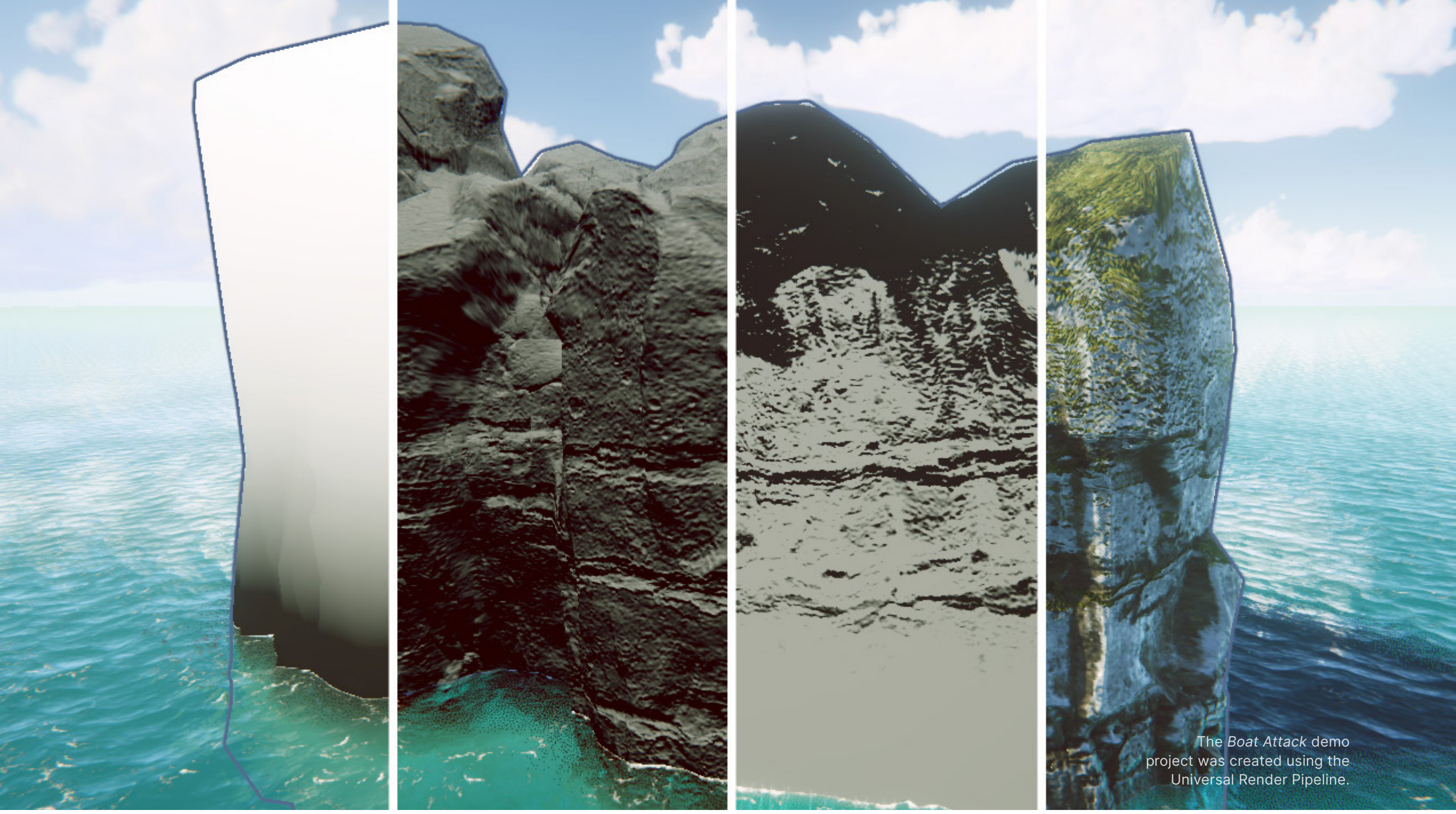
If you need to Instantiate a GameObject at runtime, a simple optimization is to parent and reposition during instantiation:

```
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.


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

Assets

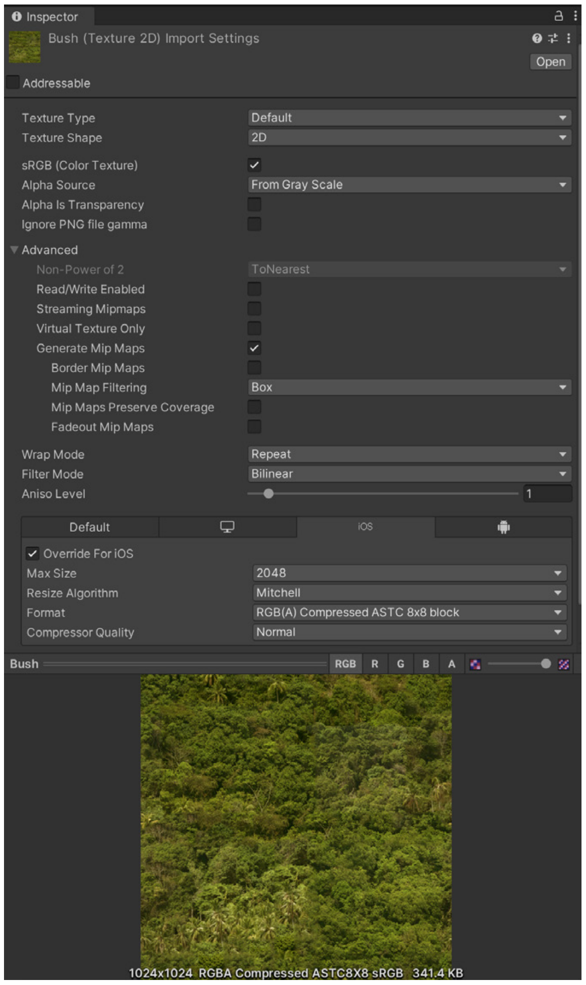
The asset pipeline can dramatically impact your application’s performance. An experienced technical artist can help your team define and enforce asset formats, specifications, and import settings.

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.

Import textures correctly

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



Bush (Texture 2D) Import Settings

Addressable

Texture Type	Default
Texture Shape	2D
sRGB (Color Texture)	✓
Alpha Source	From Gray Scale
Alpha Is Transparency	
Ignore PNG file gamma	
Advanced	
Non-Power of 2	ToNearest
Read/Write Enabled	
Streaming Mipmaps	
Virtual Texture Only	
Generate Mip Maps	✓
Border Mip Maps	
Mip Map Filtering	Box
Mip Maps Preserve Coverage	
Fadeout Mip Maps	
Wrap Mode	Repeat
Filter Mode	Bilinear
Aniso Level	1

✓ Override For iOS

Max Size	2048
Resize Algorithm	Mitchell
Format	RGB(A) Compressed ASTC 8x8 block
Compressor Quality	Normal

1024x1024 RGBA Compressed ASTC8X8 sRGB 341.4 KB

Proper texture import settings will help optimize your build size.

— **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).

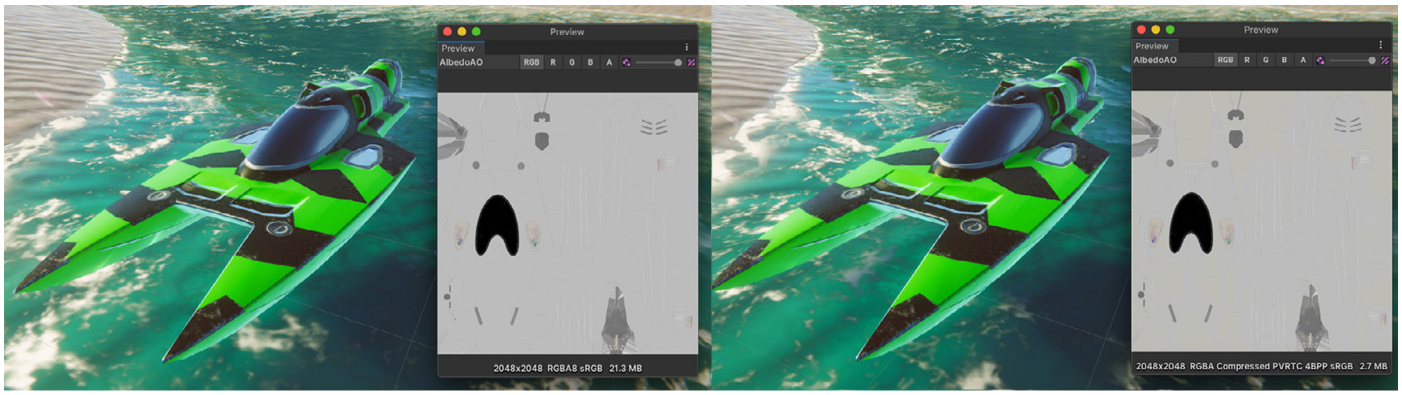
— **Atlas your textures:** Placing multiple textures into a single texture can reduce draw calls and speed up rendering. Use the Unity SpriteAtlas or the third-party Texture Packer to atlas your textures.

— **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).

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.


Preview
Preview
AlbedoAO
RGB R G B A
2048x2048 RGBA8 sRGB 21.3 MB
Preview
Preview
AlbedoAO
RGB R G B A
2048x2048 RGBA Compressed PVRTC 4BPP sRGB 2.7 MB

Uncompressed textures require more memory.

Use Adaptive Scalable Texture Compression (ATSC) for both iOS and Android. The vast majority of games in development 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)

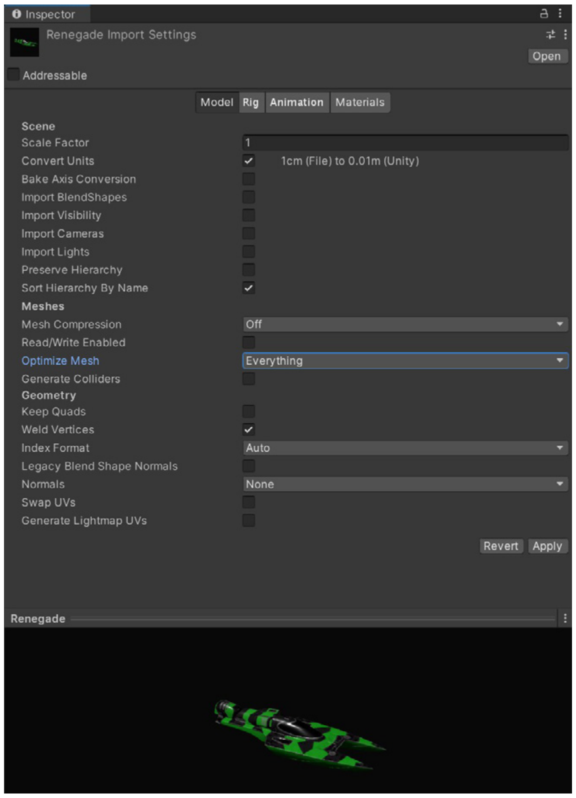
If the quality of compressed formats such as PVRTC and ETC isn't sufficiently high, and if ASTC is not fully supported on your target platform, try using 16-bit textures instead of 32-bit textures.

See the manual for more information about 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.



Addressable	
Scene	
Scale Factor	1
Convert Units	✓ 1cm (File) to 0.01m (Unity)
Bake Axis Conversion	
Import BlendShapes	
Import Visibility	
Import Cameras	
Import Lights	
Preserve Hierarchy	
Sort Hierarchy By Name	✓
Meshes	
Mesh Compression	Off
Read/Write Enabled	
Optimize Mesh	Everything
Generate Colliders	
Geometry	
Keep Quads	
Weld Vertices	✓
Index Format	Auto
Legacy Blend Shape Normals	
Normals	None
Swap UVs	
Generate Lightmap UVs	

Check your mesh import settings.

Check your polygon counts

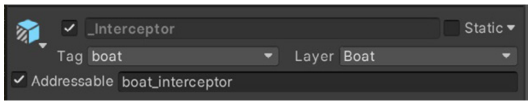
Higher-resolution models mean more memory usage and potentially longer GPU times. Does your background geometry need half a million polygons? Consider cutting down models in your DCC package of choice. Delete unseen polygons from the camera's point of view. Use textures and normal maps for fine detail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

Automate your import settings using the AssetPostprocessor

The AssetPostprocessor allows you to run scripts when importing assets. This allows you to customize settings before and/or after importing models, textures, audio, and so on.

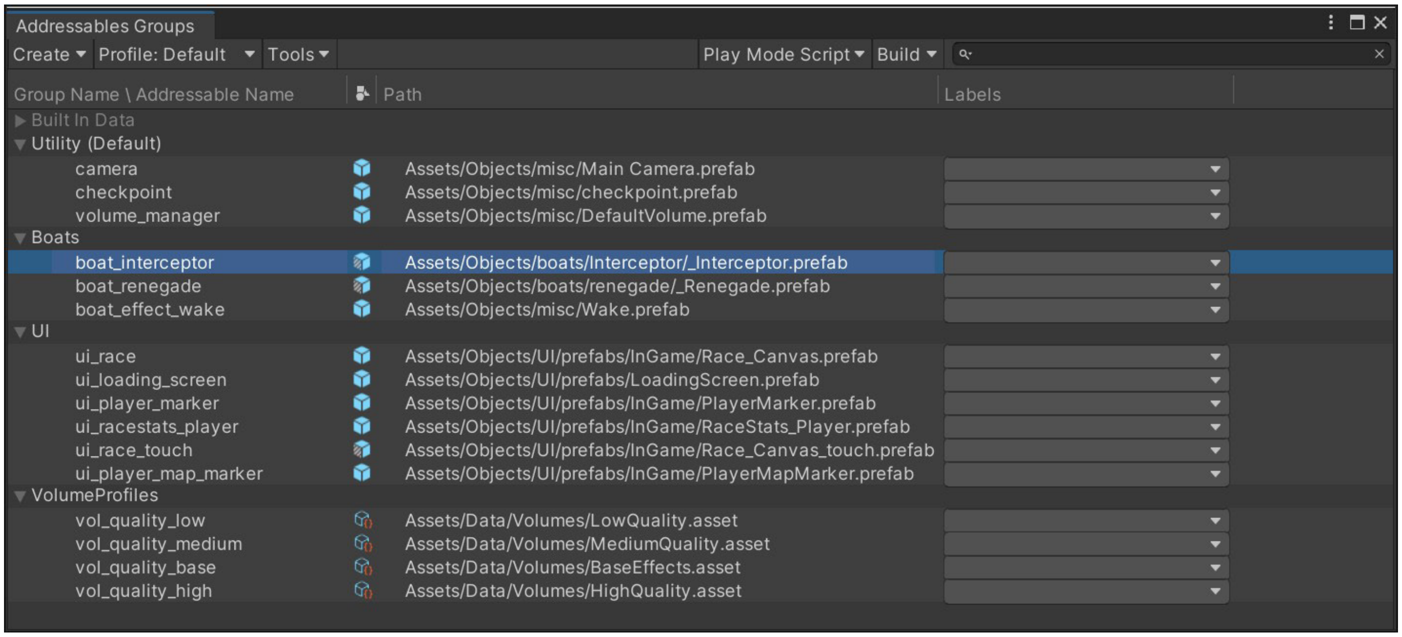
Use the Addressable Asset System

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.



Group Name \ Addressable Name	Path	Labels
Built In Data		
Utility (Default)		
camera	Assets/Objects/misc/Main Camera.prefab	
checkpoint	Assets/Objects/misc/checkpoint.prefab	
volume_manager	Assets/Objects/misc/DefaultVolume.prefab	
Boats		
boat_interceptor	Assets/Objects/boats/Interceptor/_Interceptor.prefab	
boat_renegade	Assets/Objects/boats/renegade/_Renegade.prefab	
boat_effect_wake	Assets/Objects/misc/Wake.prefab	
UI		
ui_race	Assets/Objects/UI/prefabs/InGame/Race_Canvas.prefab	
ui_loading_screen	Assets/Objects/UI/prefabs/LoadingScreen.prefab	
ui_player_marker	Assets/Objects/UI/prefabs/InGame/PlayerMarker.prefab	
ui_racestats_player	Assets/Objects/UI/prefabs/InGame/RaceStats_Player.prefab	
ui_race_touch	Assets/Objects/UI/prefabs/InGame/Race_Canvas_touch.prefab	
ui_player_map_marker	Assets/Objects/UI/prefabs/InGame/PlayerMapMarker.prefab	
VolumeProfiles		
vol_quality_low	Assets/Data/Volumes/LowQuality.asset	
vol_quality_medium	Assets/Data/Volumes/MediumQuality.asset	
vol_quality_base	Assets/Data/Volumes/BaseEffects.asset	
vol_quality_high	Assets/Data/Volumes/HighQuality.asset	

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.

Boat Attack demo project created using the Universal Render Pipeline.

Graphics and GPU optimization

Each frame, Unity determines which objects 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), while 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 you to speed up your graphics.

Batch your draw calls

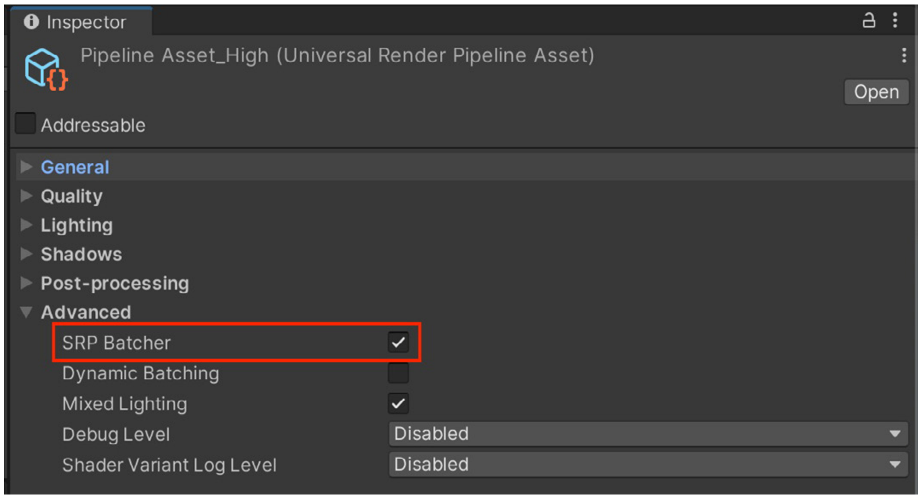
Batching objects to be drawn together minimizes the state changes needed to draw each object in a batch. This leads to improved performance by reducing the CPU cost of rendering objects. Unity can combine multiple objects into fewer batches using several techniques:

— **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 larger meshes than this, so enabling it will waste CPU time every frame looking for small meshes to batch.

— **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.

— **GPU instancing:** If you have a large number of identical objects, this technique batches them more efficiently using 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.



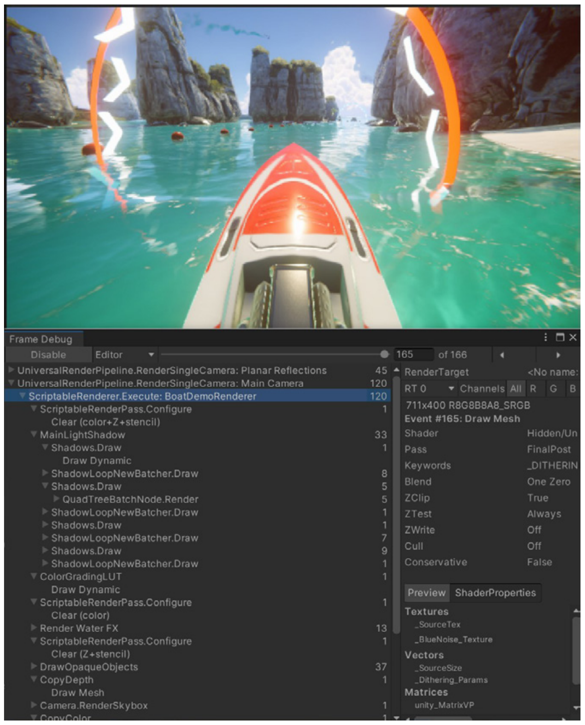
SRP Batcher ✓
Dynamic Batching
Mixed Lighting ✓
Debug Level: Disabled
Shader Variant Log Level: Disabled

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 and can help you analyze how the game is rendered.

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



```
Frame Debug
Disable   Editor ▾                                   165   of 166
▶ UniversalRenderPipeline.RenderSingleCamera: Planar Reflections   45
▼ UniversalRenderPipeline.RenderSingleCamera: Main Camera         120
  ▼ ScriptableRenderer.Execute: BoatDemoRenderer                  120
    ▼ ScriptableRenderPass.Configure                                1
        Clear (color+Z+stencil)
    ▼ MainLightShadow                                              33
      ▼ Shadows.Draw                                                1
          Draw Dynamic
      ▶ ShadowLoopNewBatcher.Draw                                   8
      ▼ Shadows.Draw                                                5
        ▶ QuadTreeBatchNode.Render                                  5
      ▶ ShadowLoopNewBatcher.Draw                                   1
      ▶ Shadows.Draw                                                1
      ▶ ShadowLoopNewBatcher.Draw                                   7
      ▶ Shadows.Draw                                                9
      ▶ ShadowLoopNewBatcher.Draw                                   1
    ▼ ColorGradingLUT                                               1
        Draw Dynamic
    ▼ ScriptableRenderPass.Configure                                1
        Clear (color)
    ▶ Render Water FX                                              13
    ▼ ScriptableRenderPass.Configure                                1
        Clear (Z+stencil)
    ▶ DrawOpaqueObjects                                            37
    ▼ CopyDepth                                                     1
        Draw Mesh
    ▶ Camera.RenderSkybox                                           1
    ▼ CopyColor                                                     1

RenderTarget              <No name:
RT 0 ▾   Channels  All  R  G  B
711x400 R8G8B8A8_SRGB
Event #165: Draw Mesh
Shader                    Hidden/Un
Pass                      FinalPost
Keywords                  _DITHERIN
Blend                     One Zero
ZClip                     True
ZTest                     Always
ZWrite                    Off
Cull                      Off
Conservative              False

Preview   ShaderProperties
Textures
  _SourceTex
  _BlueNoise_Texture
Vectors
  _SourceSize
  _Dithering_Params
Matrices
  unity_MatrixVP
```

The Frame Debugger breaks each frame into its separate steps.

Avoid too many dynamic lights

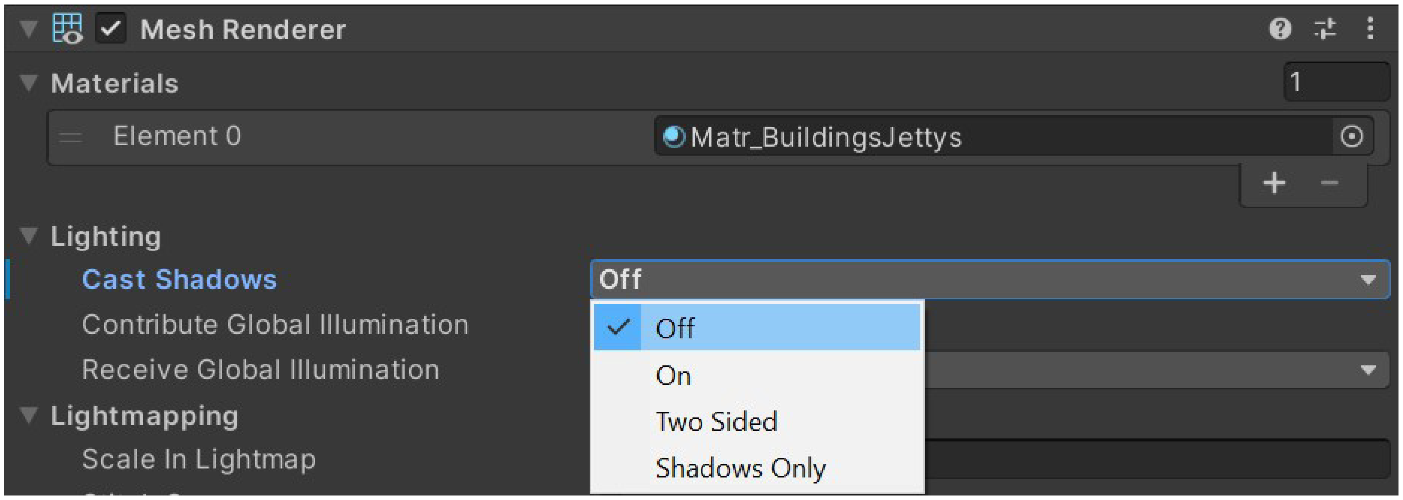
The URP reduces the number of draw calls compared to the legacy forward renderer. Avoid adding too many dynamic lights to your mobile application. Consider alternatives like custom shader effects and light probes for dynamic meshes, as well as baked lighting for static meshes.

For the specific limits of URP and Built-in Render Pipeline real-time lights, see this [feature comparison table]().

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.



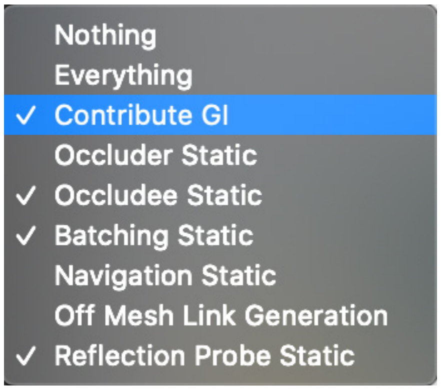
```
▼ ☑ Mesh Renderer
▼ Materials                                  1
    Element 0            Matr_BuildingsJettys
                                             +  −
▼ Lighting
    Cast Shadows                             Off ▾
    Contribute Global Illumination           ✓ Off
    Receive Global Illumination                On
▼ Lightmapping                                 Two Sided
    Scale In Lightmap                          Shadows Only
```

Disable shadow casting to reduce draw calls.

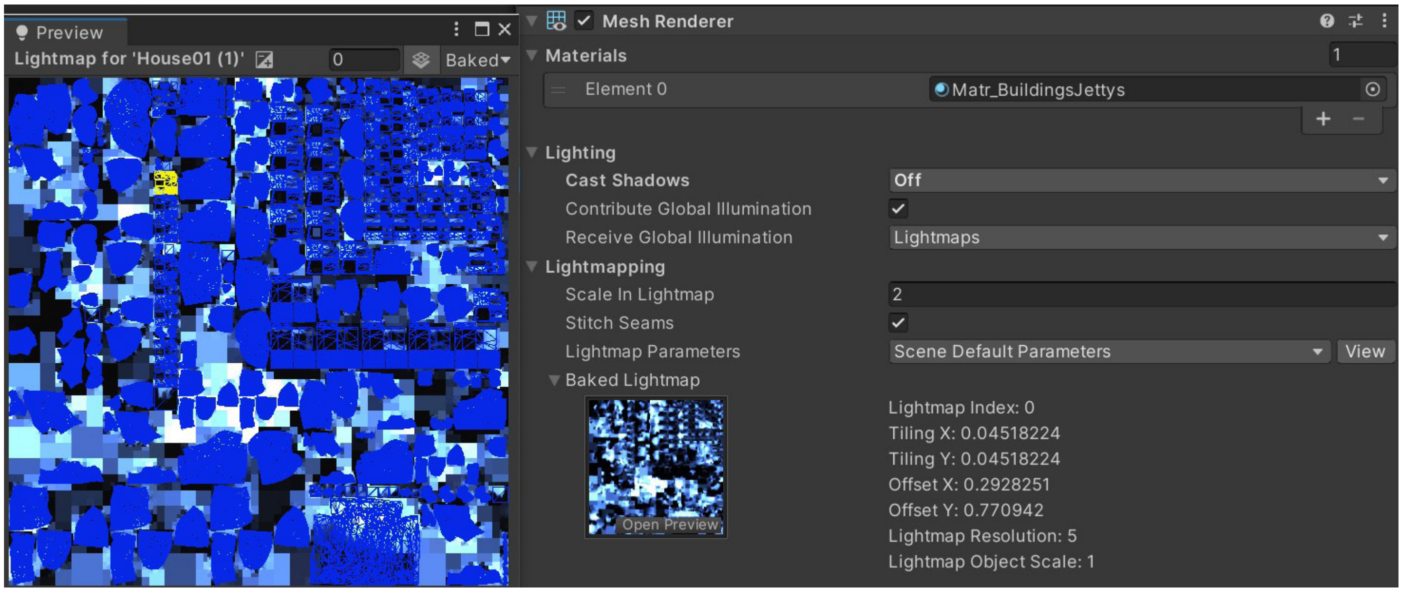
Bake your lighting into Lightmaps

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.

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.



Enable Contribute GI.



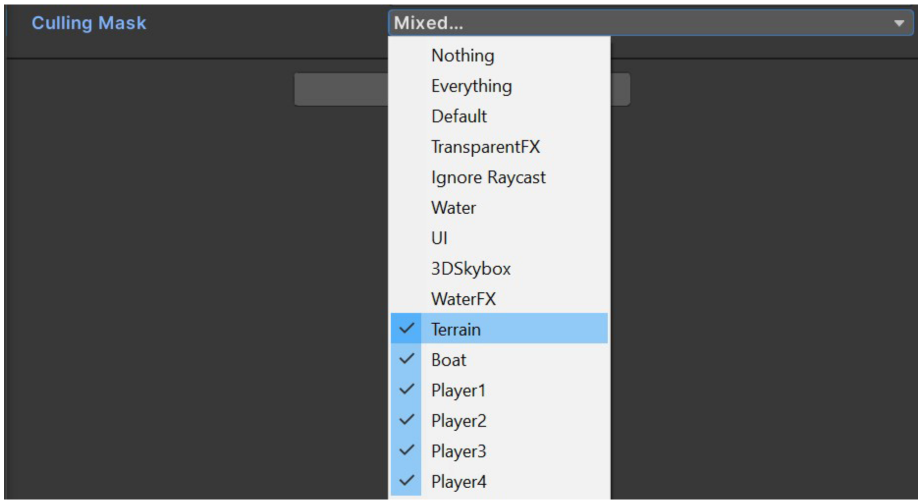
Mesh Renderer
Materials 1
Element 0 Matr_BuildingsJettys
Lighting
Cast Shadows: Off
Contribute Global Illumination: ✓
Receive Global Illumination: Lightmaps
Lightmapping
Scale In Lightmap: 2
Stitch Seams: ✓
Lightmap Parameters: Scene Default Parameters
Baked Lightmap
Lightmap Index: 0
Tiling X: 0.04518224
Tiling Y: 0.04518224
Offset X: 0.2928251
Offset Y: 0.770942
Lightmap Resolution: 5
Lightmap Object Scale: 1

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

Follow the manual guide and this article about optimizing lighting for help getting 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.

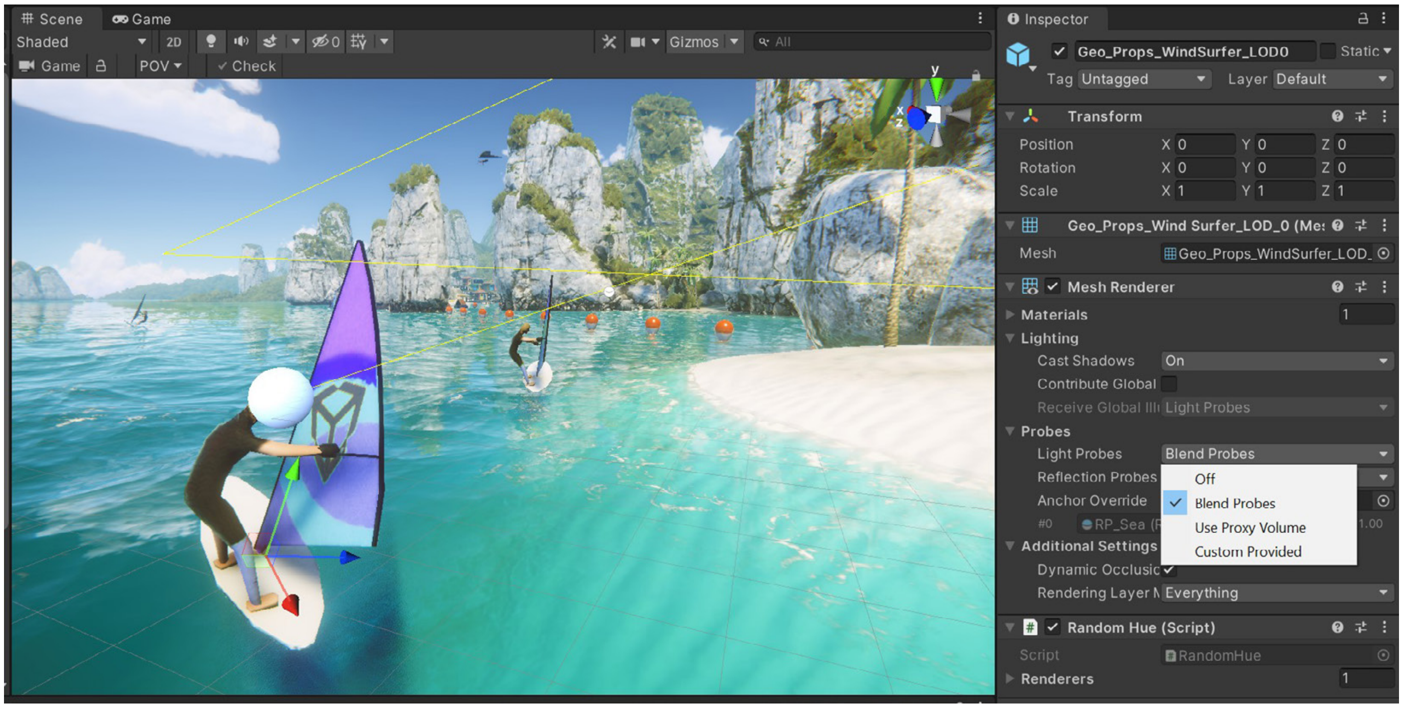


Culling Mask: Mixed...

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 and provide high-quality lighting (both direct and indirect). They use Spherical Harmonics, which calculate very quickly compared to dynamic lights.



Geo_Props_WindSurfer_LOD0
Tag: Untagged — Layer: Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Geo_Props_Wind Surfer_LOD_0 (Me:
Mesh: Geo_Props_WindSurfer_LOD_

Mesh Renderer
Materials: 1
Lighting
Cast Shadows: On
Contribute Global
Receive Global Illu: Light Probes
Probes
Light Probes: Blend Probes
Reflection Probes
Anchor Override
Additional Settings
Dynamic Occlusio ✓
Rendering Layer M: Everything

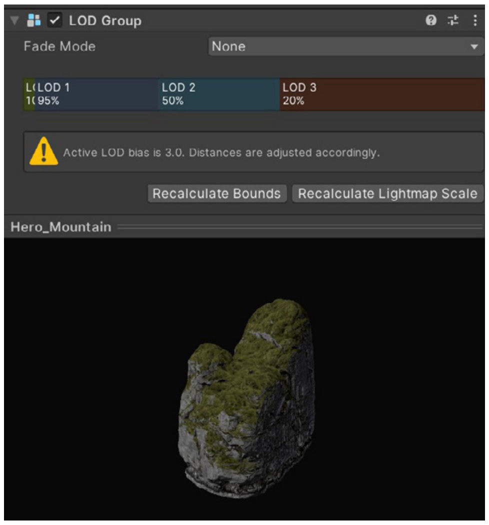
Random Hue (Script)
Script: RandomHue
Renderers: 1

Light Probes illuminate dynamic objects in the background.

Use Level of Detail (LOD)

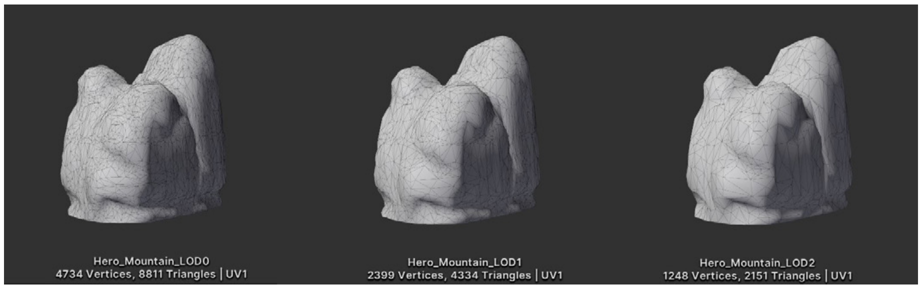
As objects move into the distance, Level of Detail can switch them to use simpler meshes with simpler materials and shaders to aid GPU performance.

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



Active LOD bias is 3.0. Distances are adjusted accordingly.

Example of a mesh using a LOD Group.


Hero_Mountain_LOD0
4734 Vertices, 8811 Triangles | UV1
Hero_Mountain_LOD1
2399 Vertices, 4334 Triangles | UV1
Hero_Mountain_LOD2
1248 Vertices, 2151 Triangles | UV1

Source meshes, modeled at varying resolutions.

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 appropriate 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

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

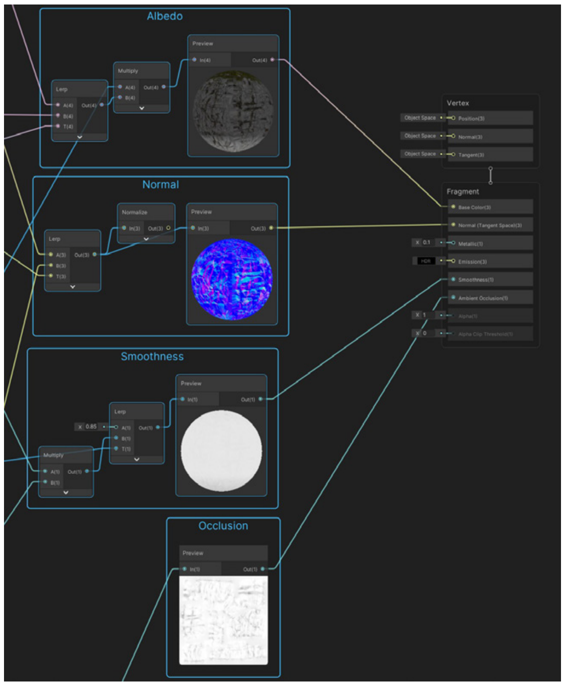
Use **Screen.SetResolution(width, height, false)** to lower the output resolution and regain some performance. Profile multiple resolutions to find the best balance between quality and speed.

Limit use of cameras

Each 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, since this 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.



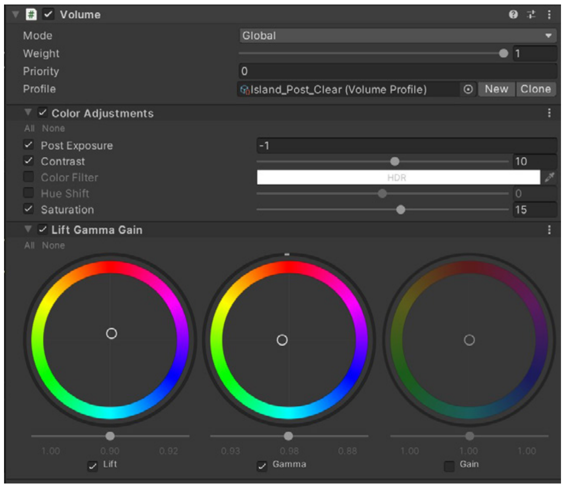
Create custom shaders with the Shader Graph.

Minimize overdraw and alpha blending

Avoid drawing unnecessary transparent or semi-transparent images. Mobile platforms are greatly impacted by the resulting overdraw and alpha blending. Don't overlap barely visible images or effects. You can check overdraw using the RenderDoc graphics debugger.

Limit Post-processing effects

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



Volume

Mode	Global
Weight	1
Priority	0
Profile	Island_Post_Clear (Volume Profile)

Color Adjustments

✓ Post Exposure	-1
✓ Contrast	10
Color Filter	HDR
Hue Shift	0
✓ Saturation	15

Lift Gamma Gain

Lift	Gamma	Gain
1.00 0.00 0.92	0.93 0.98 0.88	1.00 1.00 1.00

Keep Post-processing 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, 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.

User interface

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

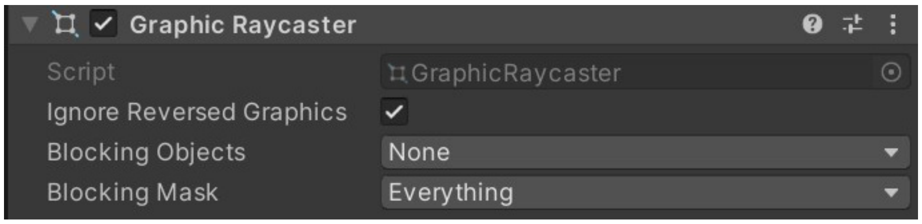
You may have UI elements that only appear sporadically in the game (e.g., a health bar that appears only when a character takes damage). If your invisible UI element is active, it might still be using draw calls. Explicitly disable any invisible UI components and re-enable them as needed.

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

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** *only* to the individual elements that need to interact (buttons, scroll rects, and so on).


Graphic Raycaster

Script	GraphicRaycaster
Ignore Reversed Graphics	✓
Blocking Objects	None
Blocking Mask	Everything

Disable Ignore Reversed Graphics, which is active by default.

Also, 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.

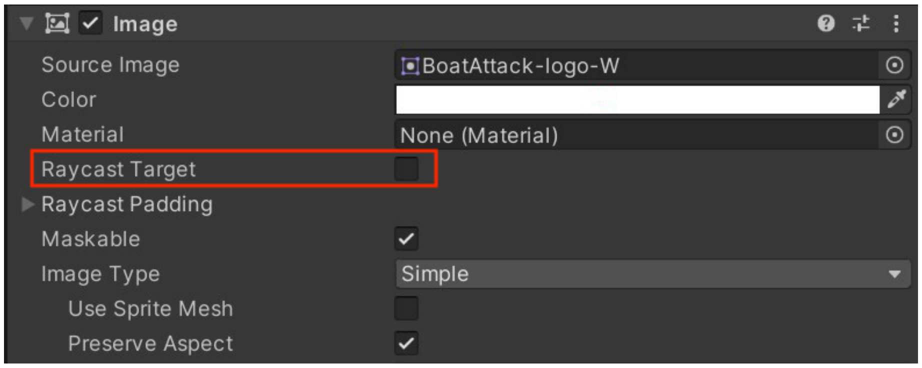


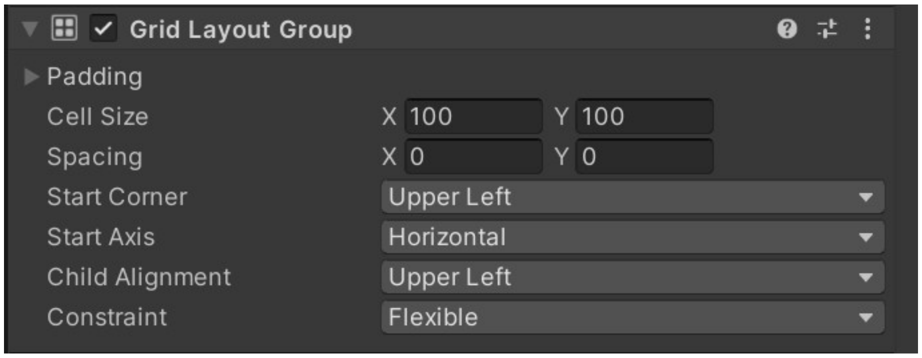
Image	
Source Image	BoatAttack-logo-W
Color	
Material	None (Material)
Raycast Target	☐
Raycast Padding	
Maskable	✓
Image Type	Simple
Use Sprite Mesh	☐
Preserve Aspect	✓

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.



Grid Layout Group	
Padding	
Cell Size	X 100 Y 100
Spacing	X 0 Y 0
Start Corner	Upper Left
Start Axis	Horizontal
Child Alignment	Upper Left
Constraint	Flexible

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.

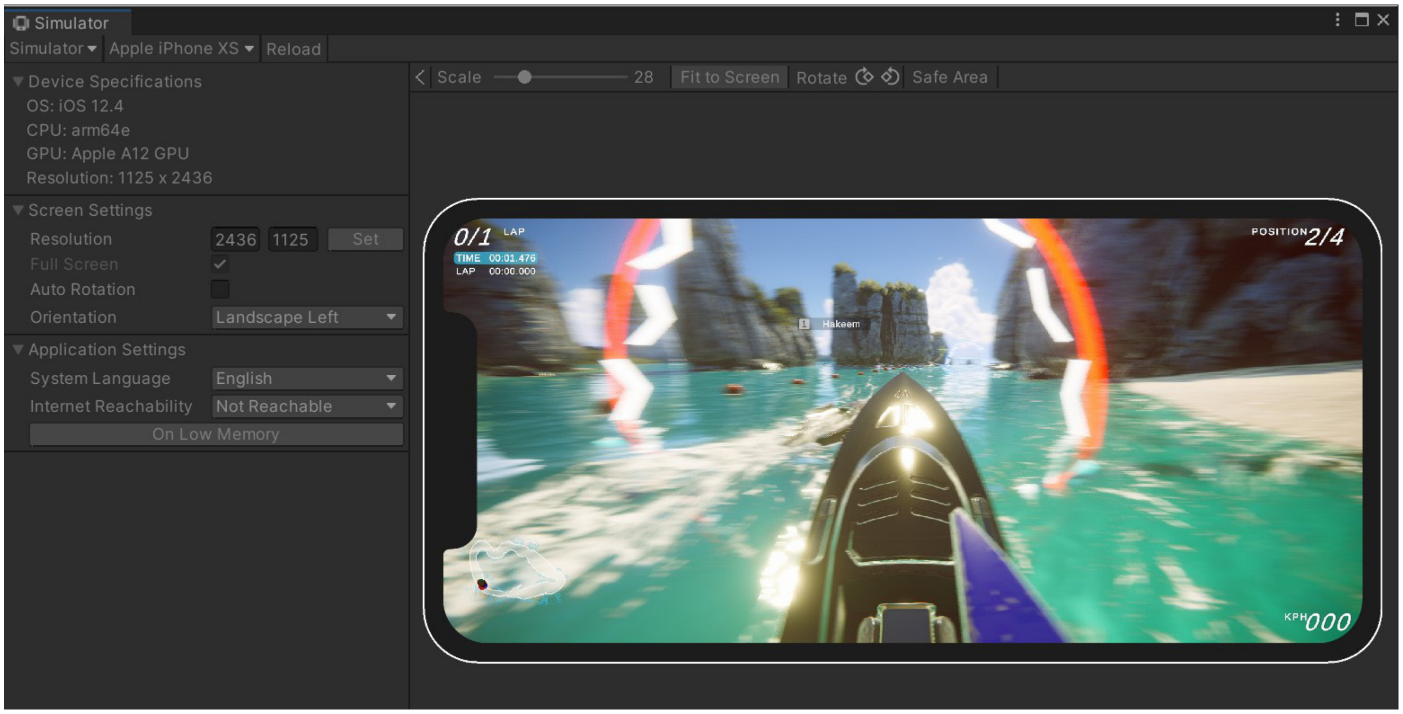
Avoid numerous overlaid elements

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.



Device Specifications
OS: iOS 12.4
CPU: arm64e
GPU: Apple A12 GPU
Resolution: 1125 x 2436

Screen Settings
Resolution 2436 1125 Set
Full Screen ✓
Auto Rotation
Orientation Landscape Left

Application Settings
System Language English
Internet Reachability Not Reachable

Preview a variety of screen formats using the Device Simulator.

When using a fullscreen UI, hide everything else

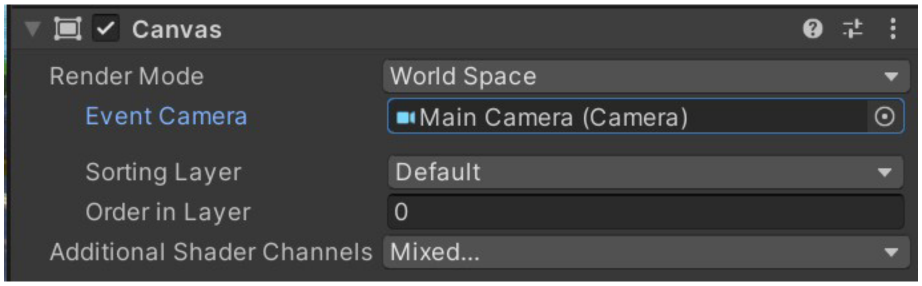
If your pause screen or start screen covers everything else in the scene, disable the camera rendering the 3D scene. Likewise, disable any background Canvas elements hidden behind the top Canvas.

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.



Canvas

Render Mode: World Space

Event Camera: Main Camera (Camera)

Sorting Layer: Default

Order in Layer: 0

Additional Shader Channels: Mixed...

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

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.

Use original uncompressed WAV files as your source assets when possible

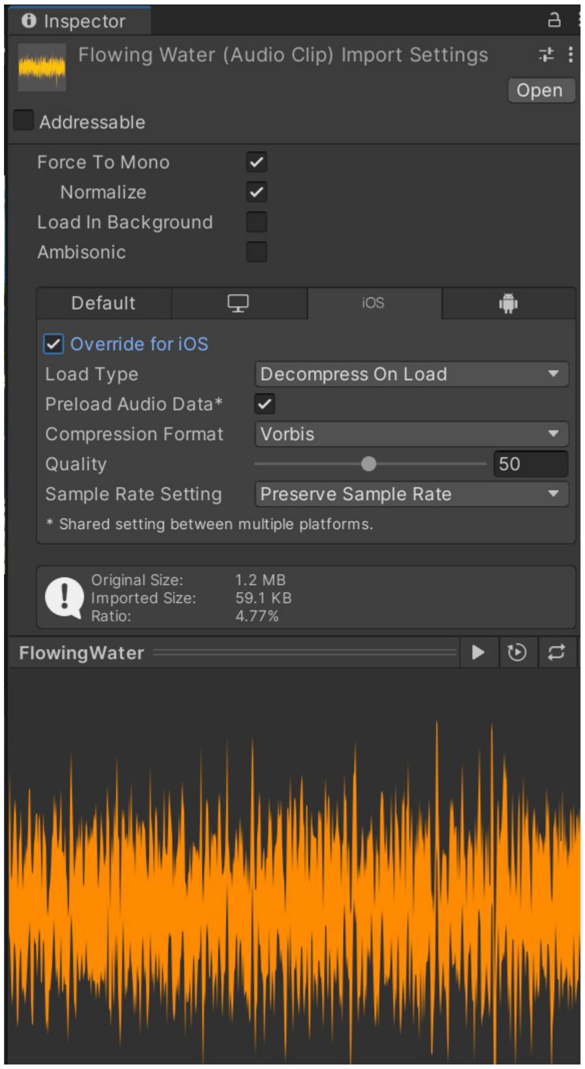
If you use any compressed format (such as MP3 or Vorbis), then Unity will decompress it and recompress it during build time. This results in two lossy passes, degrading the final quality.

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.



Addressable ☐

Force To Mono ✓
Normalize ✓
Load In Background ☐
Ambisonic ☐

✓ Override for iOS
Load Type: Decompress On Load
Preload Audio Data*: ✓
Compression Format: Vorbis
Quality: 50
Sample Rate Setting: Preserve Sample Rate
* Shared setting between multiple platforms.

Original Size: 1.2 MB
Imported Size: 59.1 KB
Ratio: 4.77%

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, but use your own ears to judge.

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

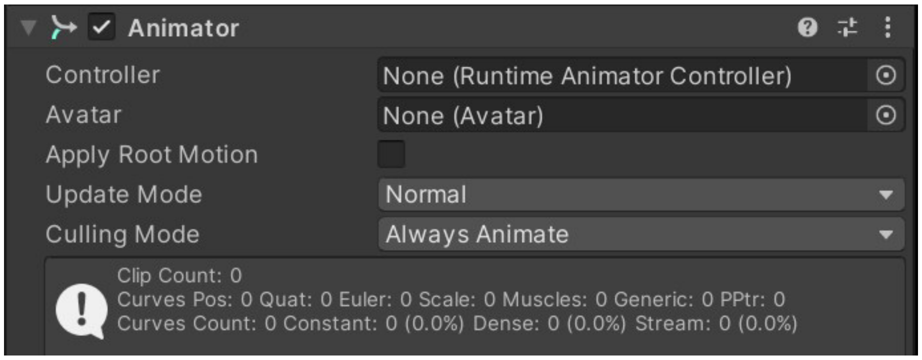
By default, Unity imports animated models with the Generic Rig, but developers often switch to the Humanoid Rig when animating a character.

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.

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).



Controller: None (Runtime Animator Controller)
Avatar: None (Avatar)
Apply Root Motion
Update Mode: Normal
Culling Mode: Always Animate

```
Clip Count: 0
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 0 Generic: 0 PPtr: 0
Curves Count: 0 Constant: 0 (0.0%) Dense: 0 (0.0%) Stream: 0 (0.0%)
```

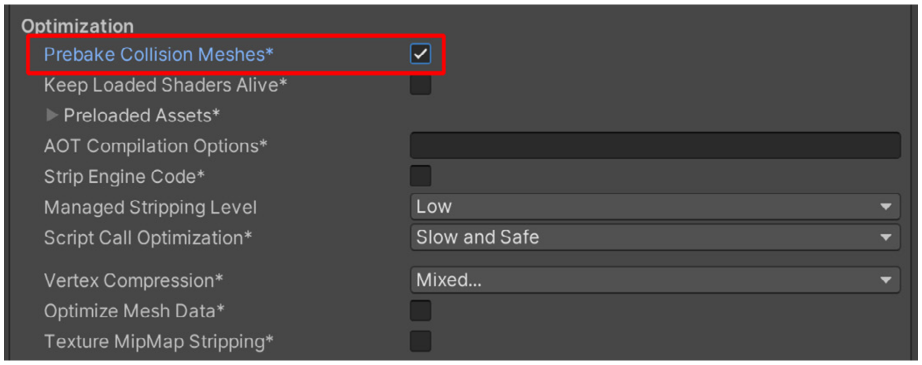
Animators are potentially expensive.

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

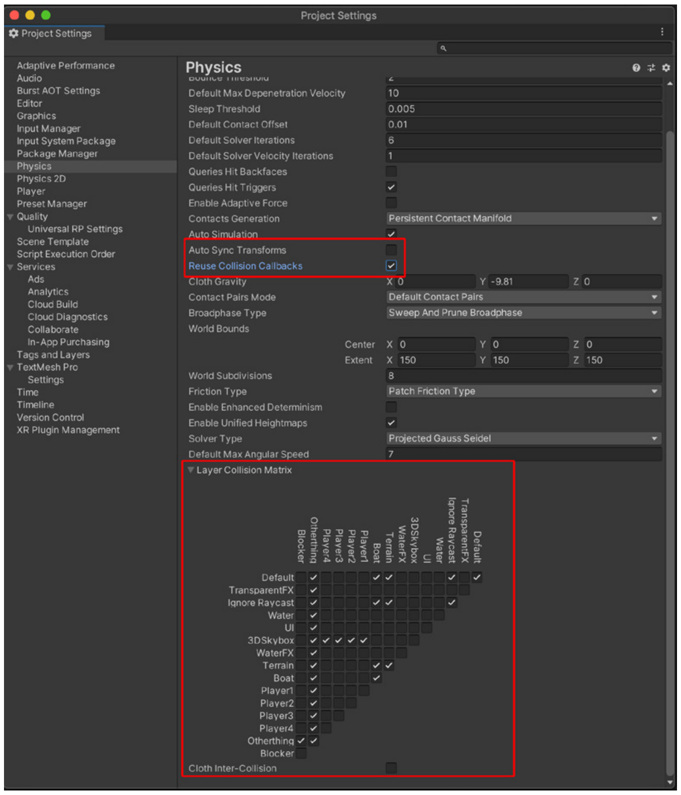
In the PlayerSettings, check **Prebake Collision Meshes** whenever possible.



Optimization

Prebake Collision Meshes*	✓
Keep Loaded Shaders Alive*	
▶ Preloaded Assets*	
AOT Compilation Options*	
Strip Engine Code*	
Managed Stripping Level	Low
Script Call Optimization*	Slow and Safe
Vertex Compression*	Mixed...
Optimize Mesh Data*	
Texture MipMap Stripping*	

Enable Prebake Collision Meshes.



Project Settings

Physics

Default Max Depenetration Velocity	10
Sleep Threshold	0.005
Default Contact Offset	0.01
Default Solver Iterations	6
Default Solver Velocity Iterations	1
Queries Hit Backfaces	
Queries Hit Triggers	✓
Enable Adaptive Force	
Contacts Generation	Persistent Contact Manifold
Auto Simulation	✓
Auto Sync Transforms	
Reuse Collision Callbacks	✓
Cloth Gravity	X 0 Y -9.81 Z 0
Contact Pairs Mode	Default Contact Pairs
Broadphase Type	Sweep And Prune Broadphase
World Bounds	
Center	X 0 Y 0 Z 0
Extent	X 150 Y 150 Z 150
World Subdivisions	8
Friction Type	Patch Friction Type
Enable Enhanced Determinism	
Enable Unified Heightmaps	✓
Solver Type	Projected Gauss Seidel
Default Max Angular Speed	7

Layer Collision Matrix

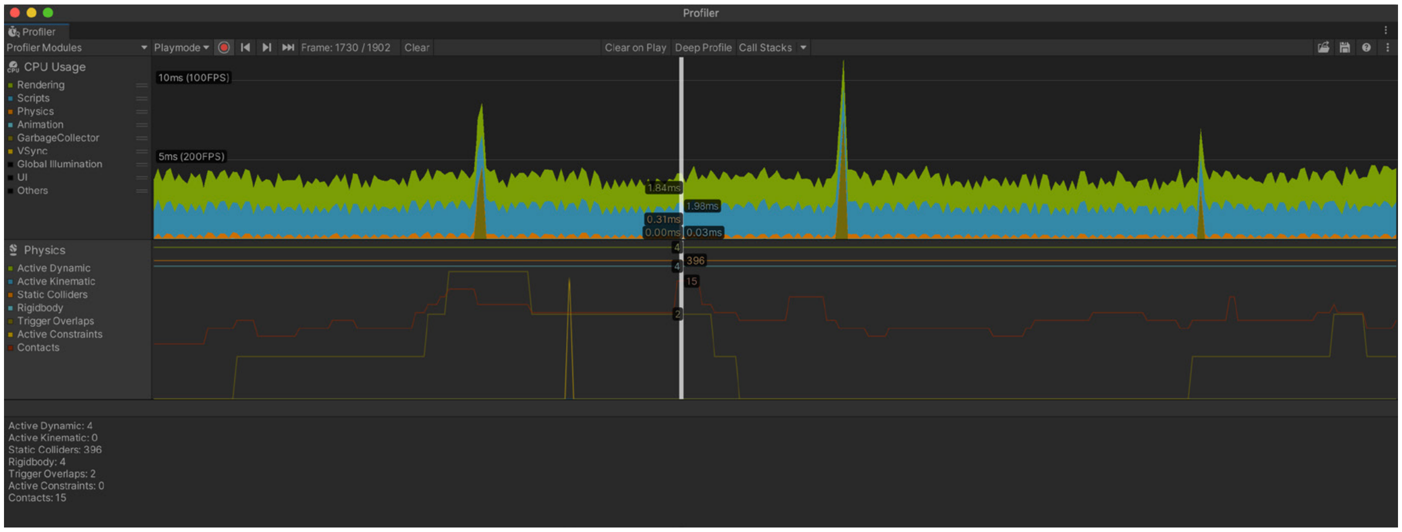
	Blocker	Otherthing	Player4	Player3	Player2	Player1	Boat	Terrain	WaterFX	3DSkybox	UI	Water	Ignore Raycast	TransparentFX	Default
Default		✓					✓	✓					✓		✓
TransparentFX		✓													
Ignore Raycast		✓					✓	✓					✓		
Water		✓													
UI		✓													
3DSkybox		✓	✓	✓	✓	✓									
WaterFX		✓													
Terrain		✓					✓	✓							
Boat		✓					✓								
Player1		✓													
Player2		✓													
Player3		✓													
Player4		✓													
Otherthing	✓	✓													
Blocker															

Cloth Inter-Collision	

Modify the physics project settings to squeeze out more performance.

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.**

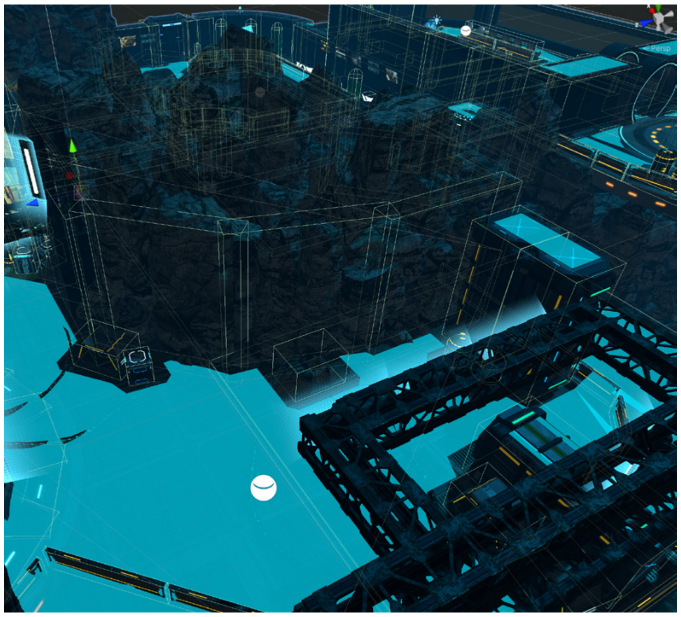


```
Active Dynamic: 4
Active Kinematic: 0
Static Colliders: 396
Rigidbody: 4
Trigger Overlaps: 2
Active Constraints: 0
Contacts: 15
```

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.



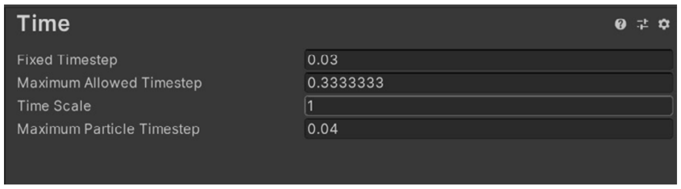
Use primitives or simplified meshes for colliders.

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).



Time

Fixed Timestep	0.03
Maximum Allowed Timestep	0.3333333
Time Scale	1
Maximum Particle Timestep	0.04

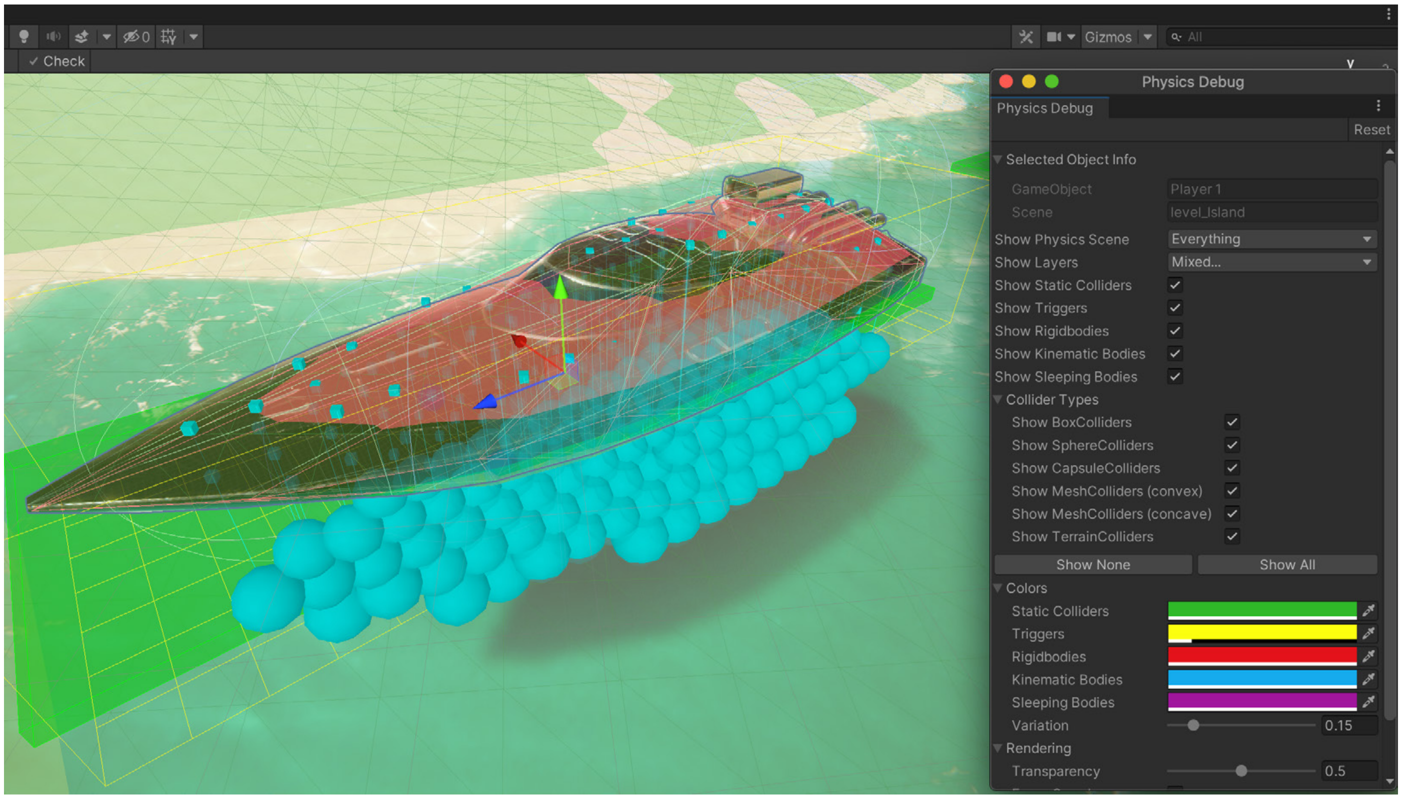
Modify the Fixed Timestep to match your target frame rate, and lower the Maximum Allowed Timestep to reduce performance glitches.

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 during a performance hitch, physics and animation may slow down, but it also reduces their impact on frame rate.

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.



Selected Object Info

GameObject	Player 1
Scene	level_Island
Show Physics Scene	Everything
Show Layers	Mixed...
Show Static Colliders	✓
Show Triggers	✓
Show Rigidbodies	✓
Show Kinematic Bodies	✓
Show Sleeping Bodies	✓

Collider Types

Show BoxColliders	✓
Show SphereColliders	✓
Show CapsuleColliders	✓
Show MeshColliders (convex)	✓
Show MeshColliders (concave)	✓
Show TerrainColliders	✓

Colors

Static Colliders	🟩
Triggers	🟨
Rigidbodies	🟥
Kinematic Bodies	🟦
Sleeping Bodies	🟪
Variation	0.15

Rendering

Transparency	0.5

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

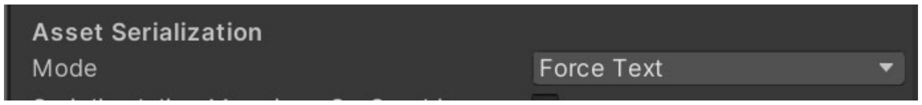
For more information, see Physics Debug Visualization in the Unity documentation.

Workflow and collaboration

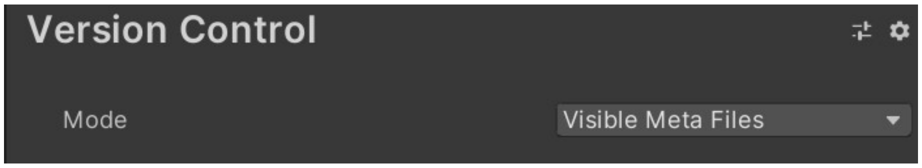
Building an application in Unity is a large endeavor that will often involve many developers. Make sure that your project is set up optimally for your team.

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**.


Asset Serialization
Mode 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**.


Version Control
Mode Visible Meta Files

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

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.

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

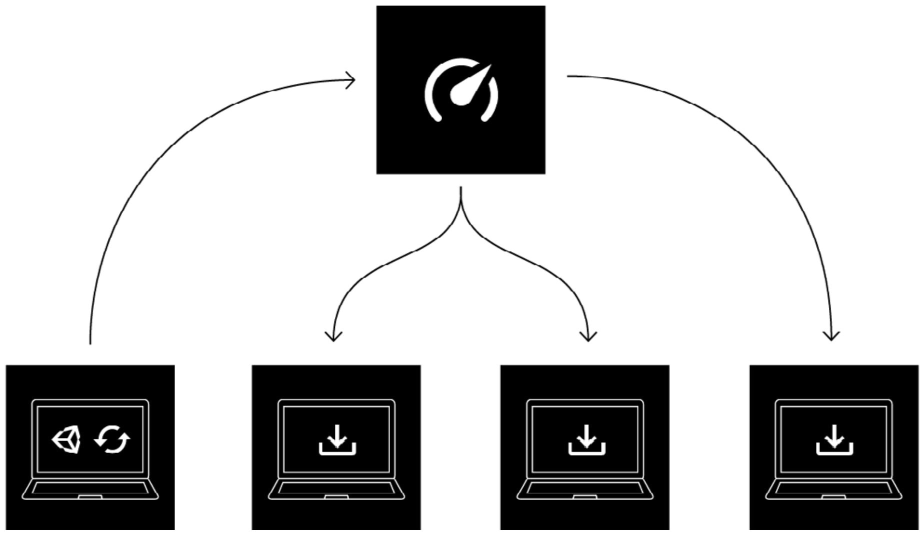
At runtime, your project can load Scenes additively using **SceneManager.LoadSceneAsync** passing the **LoadSceneMode.Additive** parameter mode.

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.

Speed Up sharing with Unity Accelerator

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



Unity Integrated Success

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

Get a Project Review

Project Reviews are an essential part of the Integrated Success package. Whenever possible, we travel to our customers and typically spend two full days familiarizing ourselves with their projects. We use various profiling tools to detect performance bottlenecks, factoring in existing requirements and design decisions. We also 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.), we perform changes while onsite and reprofile to uncover new issues.

In instances where we are unable to solve problems directly, we capture as much information as possible. Then, we conduct further investigation back at the Unity offices, consulting specialized developers across our R&D departments if need be.

Though deliverables can vary depending on the needs of the customers, typically a written report summarizes our findings and provides recommendations. Our goal is to always provide the greatest value to our customers by helping them identify potential blockers, assess risk, validate solutions and ensure that they are following best practices moving forward.

Developer Relations Manager (DRM)

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

To learn more about an Integrated Success package and Project Reviews, please reach out to your Unity sales or fill out this form.

Conclusion

You can find additional optimization tips, best practices, and news on the Unity Blog, using the **#unitytips** hashtag, on the Unity community forums, and on Unity Learn.

Performance optimization is a vast topic. 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.

unity

unity.com