



PERFORMANCE TUNING BIBLE

Written By

CyberAgent Smartphone Games &

Entertainment Division

SGE Core Technology Team



Unityパフォーマンスチューニングバイブル

株式会社サイバーエージェント | ゲーム・エンターテインメント事業部 SGEコア技術本部 著



PERFORMANCE TUNING BIBLE

Written By
CyberAgent Smartphone Games &
Entertainment Division
SGE Core Technology Team

CyberAgent.

Unityパフォーマンスチューニングバイブル

株式会社サイバーエージェント | ゲーム・エンターテインメント事業部 SGEコア技術本部 著

Introduction.

This document was created with the goal of being used as a reference when you have trouble with performance tuning of Unity applications.

Performance tuning is an area where past know-how can be utilized, and I feel that it is an area that tends to be highly individualized. Those with no experience in this field may have the impression that it is somewhat difficult. One of the reasons may be that the causes of performance degradation vary widely.

However, the workflow of performance tuning can be molded. By following that flow, it becomes easy to identify the cause of the problem and simply look for a solution that fits the event. Knowledge and experience can help in the search for a solution. Therefore, this document is designed to help you learn mainly about "workflow" and "knowledge from experience".

Although it was intended to be an internal document, we hope that many people will take the time to look at it and brush up on it. We hope that it will be of some help to those who have read it.

About this document

This manual is intended for smartphone applications. Please note that some of the explanations may not be applicable to other platforms. The version of Unity used in this document is Unity 2020.3.24f1 unless otherwise noted.

Organization of this Manual

This manual is divided into three parts. Chapter 2 covers the basics of tuning up to , Chapter 3 covers how to use various measurement tools, and Chapter 4 and beyond cover various tuning practices. The chapters are independent of each other, so you can read only what is necessary for your level. The following is an overview of each chapter.

Chapter 1 "Getting Started with Performance Tuning" describes the performance tuning workflow. First, we will discuss the preparation before starting the process, and then we will explain how to isolate the cause of the problem and proceed with

the investigation. The goal is to get you ready to start performance tuning by reading this chapter.

Chapter 2 "Fundamentals" explains the basics you should know about hardware, drawing flow, Unity mechanisms, etc. performance tuning. As you read Chapter 2 and beyond, you may want to read back when you feel your knowledge is lacking.

At Chapter 3 "Profiling Tools", you will learn how to use the various tools used for cause investigation. It is recommended to use it as a reference when using measurement tools for the first time.

The "Tuning Practice" section from Chapter 4 onward is packed with a variety of practices, from assets to scripts. Many of the contents described here can be used immediately in the field, so we encourage you to read through them.

GitHub

The repository for this book, ^{*1}, is open to the public. Additions and corrections will be made as needed. You can also point out corrections or suggest additions using PR or Issue. Please use it if you like.

Disclaimer

The information in this document is provided for informational purposes only. Therefore, any development, production, or operation using this document must be done at your own risk and discretion. We assume no responsibility whatsoever for the results of development, production, or operation based on this information.

^{*1} <https://github.com/CyberAgentGameEntertainment/UnityPerformanceTuningBible/>

Table of Contents

Introduction.	ii
About this document	ii
Organization of this Manual	ii
GitHub	iii
Disclaimer	iii
 Chapter 1 Getting Started with Performance Tuning	 1
1.1 Preliminary Preparation	2
1.1.1 Determining Indicators	3
1.1.2 Know the maximum amount of memory usage.	5
1.1.3 Decide which devices are guaranteed to work	8
1.1.4 Determine quality setting specifications.	8
1.2 Prevention	9
1.3 Work on performance tuning	10
1.3.1 Preparation for Performance Tuning	10
1.3.2 Types of Performance Degradation	11
1.4 Isolating Causes of Excess Memory	12
1.4.1 Memory leak	12
1.4.2 Memory usage is simply too high.	13
1.5 Let's investigate memory leaks.	13
1.6 Let's reduce memory	14
1.6.1 Assets	14
1.6.2 GC (Mono)	15
1.6.3 Other	16
1.6.4 Plug-ins	17
1.6.5 Examine the specifications.	17

1.7	Isolating the Cause of Processing Failures	17
1.8	Investigating Instantaneous Load	18
1.8.1	Spikes caused by GC	19
1.8.2	Spikes due to heavy processing	19
1.9	Investigate the steady-state load	19
1.9.1	CPU Bound	20
1.9.2	GPU Bound	20
1.10	Conclusion	22
Chapter 2	Fundamentals	24
2.1	Hardware	25
2.1.1	SoC	25
2.1.2	iPhone, Android and SoC	26
2.1.3	CPU	27
2.1.4	GPU	31
2.1.5	Memory	34
2.1.6	Storage	39
2.2	Rendering	42
2.2.1	Rendering Pipeline	43
2.2.2	Semi-transparent rendering and overdraw	45
2.2.3	Draw calls, set-pass calls, and batching	46
2.3	Data Representation	47
2.3.1	Bits and Bytes	48
2.3.2	Image	49
2.3.3	Image Compression	51
2.3.4	Mesh	52
2.3.5	Keyframe Animation	54
2.4	How Unity Works	55
2.4.1	Binaries and Runtime	56
2.4.2	Asset entities	60
2.4.3	Threads	61
2.4.4	Game Loop	63
2.4.5	GameObject	66
2.4.6	AssetBundle	68
2.5	C# Basics	71

Table of Contents

	2.5.1	Stack and Heap	72
	2.5.2	Garbage Collection	72
	2.5.3	Structure (struct)	74
2.6		Algorithms and computational complexity	78
	2.6.1	About computational complexity	78
	2.6.2	Basic Collections and Data Structures	81
	2.6.3	Devices to Lower the Calculation Volume	84
Chapter 3		Profiling Tools	86
	3.0.1	Points to keep in mind when measuring	87
3.1		Unity Profiler	88
	3.1.1	Measurement Methods	89
	3.1.2	CPU Usage	93
	3.1.3	Memory	98
3.2		Profile Analyzer	104
	3.2.1	How to install	105
	3.2.2	How to operate	106
	3.2.3	Analysis Result (Single mode)	106
	3.2.4	Analysis results (Compare mode)	113
3.3		Frame Debugger	113
	3.3.1	Analysis screen	114
	3.3.2	Detailed Screen	115
3.4		Memory Profiler	117
	3.4.1	How to install	118
	3.4.2	How to Operate	120
3.5		Heap Explorer	129
	3.5.1	How to install	129
	3.5.2	How to use	130
3.6		Xcode	134
	3.6.1	Profiling Methods	134
	3.6.2	Debug Navigator	135
	3.6.3	GPU Frame Capture	139
	3.6.4	Memory Graph	148
3.7		Instruments	150
	3.7.1	Time Profiler	151

	3.7.2	Allocations	153
3.8		Android Studio	156
	3.8.1	Profile Method	157
	3.8.2	CPU Measurement	158
	3.8.3	Memory Measurement	160
3.9		RenderDoc	161
	3.9.1	Measurement Method	162
	3.9.2	How to View Capture Data	165
Chapter 4		Tuning Practice - Asset	173
4.1		Texture	174
	4.1.1	Import Settings	174
	4.1.2	Read/Write	175
	4.1.3	Generate Mip Maps	176
	4.1.4	Aniso Level	176
	4.1.5	Compression Settings	177
4.2		Mesh	178
	4.2.1	Read/Write Enabled	178
	4.2.2	Vertex Compression	179
	4.2.3	Mesh Compression	180
	4.2.4	Optimize Mesh Data	181
4.3		Material	182
4.4		Animation	183
	4.4.1	Adjusting the number of skin weights	183
	4.4.2	Reducing Keys	184
	4.4.3	Reduction of update frequency	186
4.5		Particle System	187
	4.5.1	Reduce the number of particles	187
	4.5.2	Note that noise is heavy.	190
4.6		Audio	191
	4.6.1	Load Type	191
	4.6.2	Compression Format	193
	4.6.3	Sample Rate	194
	4.6.4	Set Force To Mono for sound effects.	195
4.7		Resources / StreamingAssets	195

Table of Contents

4.7.1	Resources folder slows down startup time	196
4.8	ScriptableObject	197
Chapter 5	Tuning Practice - AssetBundle	198
5.1	Granularity of AssetBundle	199
5.2	Load API for AssetBundle	199
5.3	AssetBundle unloading strategy	200
5.4	Optimization of the number of simultaneously loaded Asset-Bundles	201
Chapter 6	Tuning Practice - Physics	202
6.1	Turning Physics On and Off	203
6.2	Optimizing Fixed Timestep and Fixed Update Frequency . . .	203
6.2.1	Maximum Allowed Timestep	204
6.3	Collision Shape Selection	205
6.4	Collision Matrix and Layer Optimization	206
6.5	Raycast Optimization	207
6.5.1	Types of Raycasts	207
6.5.2	Optimization of Raycast Parameters	207
6.5.3	RaycastAll and RaycastNonAlloc	207
6.6	Collider and Rigidbody	209
6.6.1	Rigidbody and sleep states	209
6.7	Collision Detection Optimization	212
6.8	Optimization of other project settings	213
6.8.1	Physics.autoSyncTransforms	213
6.8.2	Physics.reuseCollisionCallbacks	213
Chapter 7	Tuning Practice - Graphics	214
7.1	Resolution Tuning	215
7.1.1	DPI Settings	215
7.1.2	Resolution Scaling by Script	216
7.2	Semi-transparency and overdraw	217
7.3	Reducing Draw Calls	218
7.3.1	Dynamic batching	218
7.3.2	Static batching	220
7.3.3	GPU Instancing	221

	7.3.4	SRP Batchers	224
7.4		SpriteAtlas	226
7.5		Culling	230
	7.5.1	Visual Culling	230
	7.5.2	Rear Culling	230
	7.5.3	Occlusion culling	231
7.6		Shaders	233
	7.6.1	Reducing the precision of floating-point types	234
	7.6.2	Performing Calculations with Vertex Shaders	234
	7.6.3	Prebuild information into textures	235
	7.6.4	ShaderVariantCollection	236
7.7		Lighting	238
	7.7.1	Real-time shadows	238
	7.7.2	Light Mapping	242
7.8		Level of Detail	246
7.9		Texture Streaming	247
Chapter 8		Tuning Practice - UI	249
8.1		Canvas partitioning	250
8.2		UnityWhite	251
8.3		Layout component	252
8.4		Raycast Target	253
8.5		Masks	253
8.6		TextMeshPro	254
8.7		UI Display Switching	255
Chapter 9		Tuning Practice - Script (Unity)	257
9.1		Empty Unity event functions	258
9.2		Accessing tags and names	259
9.3		Retrieving Components	259
9.4		Accessing transform	260
9.5		Classes that need to be explicitly discarded	261
9.6		String specification	261
9.7		Pitfalls of JsonUtility	262
9.8		Pitfalls of Render and MeshFilter	263

Table of Contents

9.9	Removal of log output codes	264
9.10	Accelerate your code with Burst	265
9.10.1	Using Burst to Speed Up Code	266
Chapter 10	Tuning Practice - Script (C#)	269
10.1	GC.Alloc cases and how to deal with them	270
10.1.1	New of reference type	270
10.1.2	Lambda Expressions	271
10.1.3	Cases where generics are used and boxed	273
10.2	About for/foreach	275
10.3	Object Pooling	278
10.4	string	279
10.5	LINQ and Latency Evaluation	281
10.6	How to avoid async/await overhead	285
10.7	Optimization with stackalloc	287
10.8	Optimizing method invocation under IL2CPP backend with sealed	288
10.9	Optimization through inlining	291
Chapter 11	Tuning Practice - Player Settings	293
11.1	Scripting Backend	294
11.1.1	Debug	295
11.1.2	Release	295
11.1.3	Master	295
11.2	Strip Engine Code / Managed Stripping Level	295
11.3	Accelerometer Frequency (iOS)	296
Chapter 12	Tuning Practice - Third Party	297
12.1	DOTween	298
12.1.1	SetAutoKill	298
12.1.2	SetLink	299
12.1.3	DOTween Inspector	300
12.2	UniRx	301
12.2.1	Unsubscribe	301
12.3	UniTask	302
12.3.1	UniTask v2	302

12.3.2 UniTask Tracker	302
CONCLUSION	305
Introduction of the Authors	306



PERFORMANCE TUNING BIBLE

CHAPTER

01

第1章

**Getting Started with
Performance Tuning**

CyberAgent Smartphone Games & Entertainment

Chapter 1

Getting Started with Performance Tuning

This chapter describes the preparation required for performance tuning and the flow of the process.

First, we will cover what you need to decide on and consider before starting performance tuning. If your project is still in the early stages, please take a look at it. Even if your project is somewhat advanced, it is a good idea to check again to see if you have taken into account the information listed in this section. Next, we will explain how to address the application that is experiencing performance degradation. By learning how to isolate the cause of the problem and how to resolve it, you will be able to implement a series of performance tuning flows.

1.1 Preliminary Preparation

Before performance tuning, decide on the indicators you want to achieve. It is easy to say in words, but it is actually a highly challenging task. This is because the world is full of devices with various specifications, and it is impossible to ignore users with low-specification devices. Under such circumstances, it is necessary to consider various factors, such as game specifications, target user groups, and whether or not the game will be developed overseas. This work cannot be completed by engineers alone. It is necessary to determine quality lines in consultation with people in other professions, and technical verification will also be necessary.

It is highly difficult to determine these indicators from the initial phase when there are not enough function implementations or assets to measure the load. Therefore, one approach is to determine them after the project has progressed to a certain degree. However, it is important to **make sure that the decision is made before the project enters the mass production phase**. This is because once mass production

is started, the cost of change will be enormous. It will take time to decide on the indicators introduced in this section, but do not be in a hurry and proceed firmly.

Fear of specification changes after the mass production phase

Suppose you have a project that is now in the post-production phase, but has a rendering bottleneck on a low-spec terminal. Memory usage is already near its limit, so switching to a lower-load model based on distance is not an option. Therefore, we decide to reduce the number of vertices in the model.

First, we will reorder the data for reduction. A new purchase order will be needed. Next, the director needs to check the quality again. And finally, we also need to debug. This is a simplified description, but in reality there will be more detailed operations and scheduling.

After mass production, there will be dozens to hundreds of assets that will need to be handled as described above. This is time-consuming and labor-intensive, and can be fatal to the project.

To prevent such a situation, it is very important to **create the most burdensome scenes** and **verify in advance** whether they meet the indicators.

1.1.1 Determining Indicators

Determining indicators will help you determine the goals to be achieved. On the other hand, if there are no indicators, the project will never end. Table 1.1 The following is a list of indicators that you should decide on.

▼ Table 1.1 Indicators

Item	Element
Frame rate	How much frame rate to aim for at all times.
Memory	Estimate the maximum memory on which screen and determine the limit value.
Transition time	How long is the appropriate transition time wait?
Heat	How much heat can be tolerated in X hours of continuous play
Battery	How much battery consumption is acceptable for X hours of continuous play

Table 1.1 **Frame rate and memory** are the most important indicators among the

Chapter 1 Getting Started with Performance Tuning

above, so be sure to decide on them. At this point, let's leave low-specification devices out of the equation. First of all, it is important to determine the indicators for devices in the volume zone.

The definition of the volume zone depends on the project. You may want to decide based on market research or other titles that can be used as benchmarks. Or, given the background of prolonged replacement of mobile devices, you may use the mid-range of about four years ago as a benchmark for now. Even if the rationale is a bit vague, let's set a flag to aim for. From there, you can make adjustments.

Let us consider an actual example. Suppose you have a project with the following goals

- We want to improve everything that is wrong with our competitor's application.
- We want to make it run smoothly, especially ingame.
- Other than the above, we want to be as good as the competition.

When the team verbalized these vague goals, the following metrics were generated.

- Frame rate
 - 60 frames ingame and 30 frames outgame from a battery consumption perspective.
- Memory
 - To speed up the transition time, the design should retain some out-game resources during ingame. The maximum amount of memory used shall be 1 GB.
- Transition Time
 - Transition time to ingame and outgame should be at the same level as the competition. In time, it should be within 3 seconds.
- Heat
 - Same level as the competition. It does not get hot for 1 hour continuously on the verified device. (Not charging)
- Battery

- Same level as competitors. Battery consumption is about 20% after 1 hour of continuous use on the tested device.

Once you have determined the target, you can use a reference device to test it. If the target is not reached at all, it is a good indicator.

Optimization by game genre

In this case, the theme of the game was to run smoothly, so the frame rate was set at 60 frames per second. A high frame rate is also desirable for rhythm action games and games with severe judgments such as first-person shooters (FPS). However, there is a disadvantage to a high frame rate. The higher the frame rate, the more battery power is consumed. In addition, the more memory is used, the more likely it is to be killed by the OS when it suspends. Considering these advantages and disadvantages, decide on an appropriate target for each game genre.

1.1.2 Know the maximum amount of memory usage.

This section focuses on maximum memory usage. To determine the maximum memory usage, first determine how much memory is available on the supported devices. Basically, it is a good idea to verify this with the lowest specification device that is guaranteed to work. However, since the memory allocation mechanism may have changed depending on the OS version, it is recommended to prepare multiple devices with different major versions if possible. Also, since the measurement logic differs depending on the measurement tool, be sure to use only one tool.

For reference, the following is a description of the verification conducted by the author on iOS. In the verification project, Texture2D was generated at runtime, and the time required for a crash was measured. The code is as follows

▼ List 1.1 Verification code

Chapter 1 Getting Started with Performance Tuning

```
1: private List<Texture2D> _textureList = new List<Texture2D>();
2: ...
3: public void CreateTexture(int size) {
4:     Texture2D texture = new Texture2D(size, size, TextureFormat.RGBA32, false);
5:     _textureList.Add(texture);
6: }
```

The result of the verification is as follows: Figure 1.1

Devices	Memory (GB)	OS	Memory Usage (GB)
iPhone6	1	12.4.1	0.65
iPhone6S	2	10.0.1	1.37
		11.3	2.61
		12.1.2	1.37
		13.6	1.42
iPhone7	2	10.3.1	1.31
		11	2.64
		12.4	1.37
		13.3.1	1.42
iPhone7 Plus	3	12.0.1	2.00
iPhoneX	3	12.1	1.76
iPhoneXR	3	13.5.1	1.81

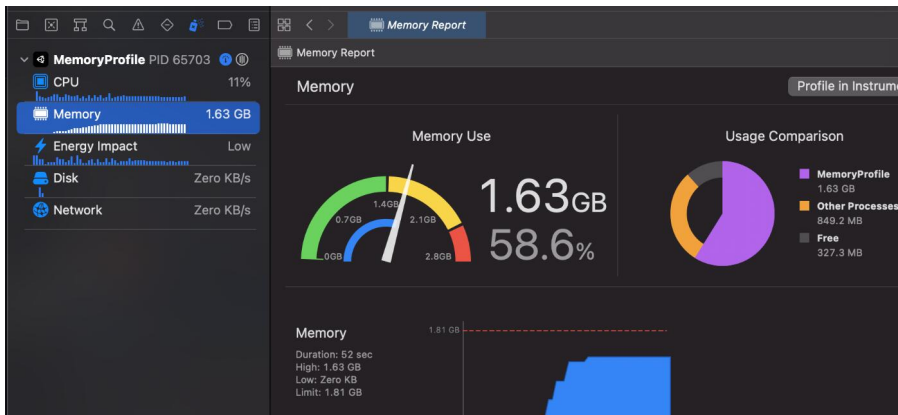
▲ Figure 1.1 Crash Threshold.

The verification environment is Unity 2019.4.3 and Xcode 11.6, using the values in the Memory section of Xcode's Debug Navigator as reference. Based on the results of this verification, it is recommended to keep the memory within 1.3 GB for devices with 2 GB of onboard memory, such as the iPhone 6S and 7. It can also be seen that when supporting devices with 1GB of onboard memory, such as the iPhone 6, the memory usage constraints are much stricter. Another characteristic of iOS11 is that its memory usage is significantly higher than that of the iPhone 6, possibly due to a different memory management mechanism. When verifying, please note that such differences due to operating systems are rare.

Figure 1.1 In the following example, the test environment is a little old, so some of the measurements have been re-measured using the latest environment at the time of writing. We used Unity 2020.3.25 and 2021.2.0 and Xcode 13.3.1 to build on iPhoneXR with OS versions 14.6 and 15.4.1. As a result, there was no particular difference in the measured values, so I think the data is still reliable.

Memory measurement tools

We recommend native-compliant tools such as Xcode and AndroidStudio for memory measurement. For example, the Unity Profiler does not measure native memory allocated by plug-ins. In the case of IL2CPP builds, IL2CPP metadata (about 100MB) is also not included in the measurement. On the other hand, in the case of the native tool Xcode, all memory allocated by the application is measured. Therefore, it is better to use a native-compliant tool that measures values more accurately.



▲ Figure 1.2 Xcode Debug Navigator

1.1.3 Decide which devices are guaranteed to work

It is also important to decide on the minimum guaranteed terminal as an indicator to determine how far to go in performance tuning. It is difficult to decide on a guaranteed device immediately without experience, but do not decide on a spur-of-the-moment basis, but rather start by identifying candidates for low-specification devices.

The method I recommend is to refer to the data measured by "SoC specs". Specifically, look for data measured by benchmark measurement applications on the Web. First, you need to know the specifications of the device you use as a reference, and then select a few devices with a somewhat lower measured value.

Once you have identified the devices, actually install the application and check its operation. Do not be discouraged if the operation is slow. You are now at the starting line where you can discuss what to eliminate. In the next section, we will introduce some important specifications that you should consider when eliminating features.

There are several benchmark measurement applications, but I use Antutu as my benchmark. This is because there is a website that compiles measurement data and volunteers are actively reporting their measurement data.

1.1.4 Determine quality setting specifications.

With the market flooded with devices of various specifications, it would be difficult to cover many devices with a single specification. Therefore, in recent years, it has become common practice to set several quality settings in the game to guarantee stable operation on a variety of devices.

For example, the following items can be classified into high, medium, and low quality settings.

- Screen resolution
- Number of objects displayed
- Shadows
- Post-effect function

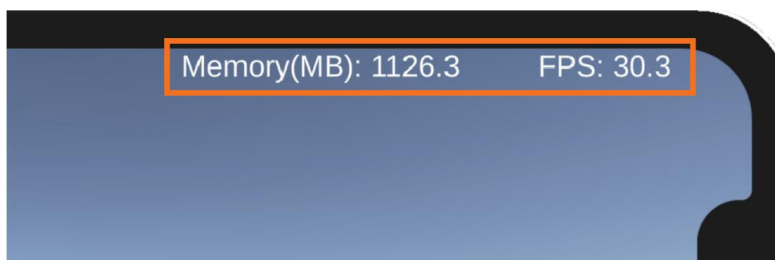
- Frame rate
- Ability to skip CPU-intensive scripts, etc.

However, this will reduce the quality of the look and feel of the project, so please consult with the director and together explore what line is acceptable for the project.

1.2 Prevention

As with defects, performance degradation can have a variety of causes over time, increasing the difficulty of investigation. It is a good idea to implement a mechanism in your application that will allow you to notice the problem as early as possible. A simple and effective way to do this is to display the current application status on the screen. It is recommended that at least the following elements be displayed on the screen at all times

- Current frame rate
- Current memory usage



▲ Figure 1.3 Performance Visualization

While frame rate can be detected by the user's experience that performance is declining, memory can only be detected by crashes. Figure 1.3 The probability of detecting memory leaks at an early stage will increase simply by constantly displaying them on the screen as shown in the following table.

This display method can be further improved to be more effective. For example, if the target frame rate is 30 frames per second, turn the display green for frames between 25 and 30, yellow for frames between 20 and 25, and red for frames below that. This way, you can intuitively see at a glance whether the application meets the

criteria.

1.3 Work on performance tuning

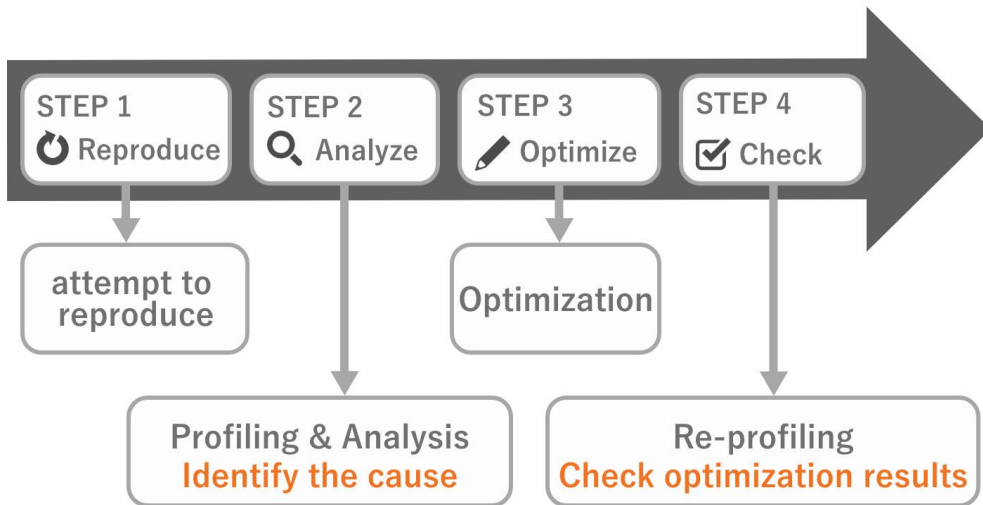
No matter how much effort you put into preventing performance degradation before it happens, it will be tough to prevent it all. This is unavoidable. Performance degradation is an inseparable part of development. There will come a time when you will have to face performance tuning. In the following sections, we will explain how you should tackle performance tuning.

1.3.1 Preparation for Performance Tuning

Before starting performance tuning, let's first introduce an important attitude. For example, let's say you have an application with a slow frame rate. Obviously, several rich models are displayed. People around you are saying that these models must be the cause. We need to look carefully at the evidence to see if this is really the case. There are two things to keep in mind when tuning performance.

The first is to **measure and identify the cause. Do not guess.**

Second, after making corrections, be sure to **compare the results.** You may want to compare the before and after profiles. The point is to check for performance degradation across the board, not just in the modified area. The scary part of performance tuning is that in rare cases, the modified part is faster, but the load increases in other parts of the system, and the overall performance is degraded. This is the end of the line.

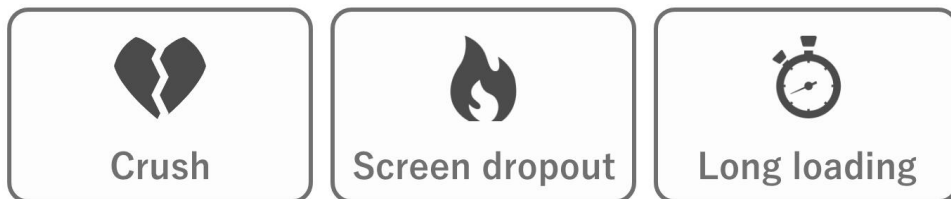


▲ Figure 1.4 Preparation for Performance Tuning

To identify the cause of the problem and confirm that the system has become faster. This is an important attitude for performance tuning.

1.3.2 Types of Performance Degradation

Performance degradation may refer to different things. In this document, we define the three broad categories as follows. (Figure 1.5)



▲ Figure 1.5 Causes of performance degradation

First, crashes can be classified into two main types: **"memory overflow"** or **"program execution error"**. The latter is not the domain of performance tuning, so the specifics will not be covered in this document.

Next, **"CPU and GPU processing time"** will probably account for the majority of

screen dropouts and long loading times. In the following sections, we will focus on "memory" and "processing time" to delve deeper into performance degradation.

1.4 Isolating Causes of Excess Memory

We have already discussed possible causes of crashes due to excess memory. From here, let's further break down the causes of excess memory.

1.4.1 Memory leak

One possible cause of memory overflow is a memory leak. To check for this, let's see if memory usage gradually increases with scene transitions. Scene transitions here are not just screen transitions, but also large screen changes. For example, from the title screen to out-game, from out-game to in-game, etc. Follow the steps below to measure the memory usage.

1. Note the memory usage in a certain scene
2. Transition to a different scene
3. Repeat "1" to "2" about 3 to 5 times

If the measurement results show a net increase in memory usage, something is definitely leaking. This can be called an invisible defect. First, let's eliminate the leak.

It is also a good idea to sandwich a few screen transitions before making the "2" transition. This is because it is possible that only the resources loaded on a particular screen are exceptionally leaked.

Once you are sure of the leak, you should look for the cause of the leak. "1.5 Let's investigate memory leaks." explains how to investigate specifically.

Reasons for repeating

This is the author's experience, but there were cases where some resources were not released due to timing issues after resource release (after `UnloadUnusedAssets`). These unreleased resources are released when transitioning to the next scene. In contrast, a gradual increase in memory usage with repeated transitions will eventually cause a crash. In order to separate

1.5 Let's investigate memory leaks.

the former problem from the latter, this document recommends repeating transitions several times during memory measurement.

Incidentally, if there is a problem like the former, some object is probably still holding a reference at the time of resource release and is subsequently released. It is not fatal, but it is a good idea to investigate the cause of the problem and resolve it.

1.4.2 Memory usage is simply too high.

If memory usage is high without leaks, it is necessary to explore areas where it can be reduced. Specific methods are described at "1.6 Let's reduce memory".

1.5 Let's investigate memory leaks.

First, let's reproduce the memory leak and then use the following tools to find the cause. Here we will briefly explain the features of the tools. Details on how to use the tools are handled at Chapter 3 "Profiling Tools", so please refer to it while investigating.

Profiler (Memory)

This is a profiler tool that is included by default in the Unity editor. Therefore, you can easily perform measurement. Basically, you should snapshot the memory with "Detailed" and "Gather object references" set and investigate. Unlike other tools, this tool does not allow snapshot comparisons of measurement data. For more information on how to use this tool, please refer to "3.1.3 Memory".

Memory Profiler

This one must be installed from Package Manager. It graphically displays memory contents in a tree map. It is officially supported by Unity and is still being updated frequently. Since v0.5, the method of tracking reference relationships has been greatly improved, so we recommend using the latest version. See "3.4 Memory Profiler" for details on usage.

Heap Explorer

This must be installed from the Package Manager. It is a tool developed by an individual, but it is very easy to use and lightweight. It is a tool that can be used to track references in a list format, which is a nice addition to the Memory Profiler of v0.4 and earlier. It is a good alternative tool to use when the v0.5 Memory Profiler is not available. See "3.5 Heap Explorer" for details on how to use it.

1.6 Let's reduce memory

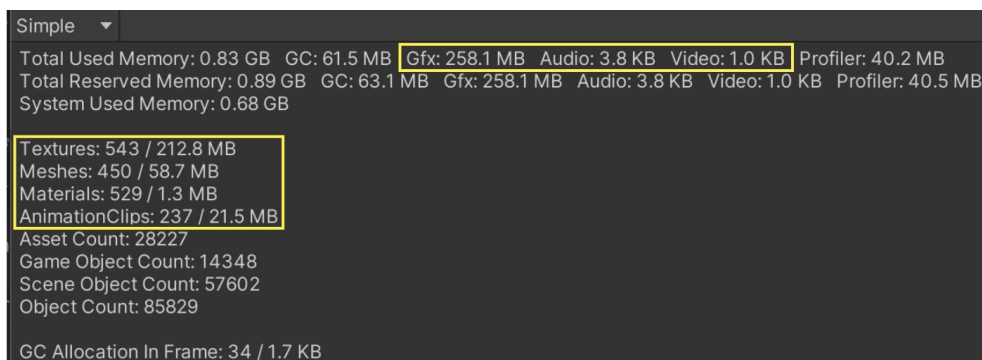
The key to reducing memory is to **cut from large areas**. Because 1,000 pieces of 1KB will only result in a 1MB reduction. However, if you compress a 10 MB texture to 2 MB, you can reduce it by 8 MB. Considering cost-effectiveness, be aware that you should start with the largest items and reduce them first.

In this section, the tool used for memory reduction is Profiler(Memory). If you have never used it, please visit "3.1.3 Memory" for more information.

Subsequent sections will cover items to look for when reducing.

1.6.1 Assets

If Simple View has a lot of Assets, it may be due to unnecessary assets or memory leaks. The Assets-related area here is the area enclosed by the rectangle at Figure 1.6.



▲ Figure 1.6 Assets-related items

In this case, there are three things to investigate

Unnecessary Assets Investigation

Unnecessary assets are resources that are not needed at all for the current scene. For example, background music that is only used in the title screen is residing in memory even in out game. First, make sure that only those assets that are necessary for the current scene are used.

Duplicate Asset Investigation

This often occurs when supporting asset bundles. The same asset is included in multiple asset bundles due to poor separation of asset bundle dependencies. However, too much separation of dependencies leads to an increase in the number of downloaded files and the cost of file deployment. It may be necessary to develop a sense of balance while measuring this area. For more information on asset bundling, please refer to "2.4.6 AssetBundle".

Check the Regulations

Review each item to see if the regulations are followed. If there are no regulations, check to see if you may not be estimating memory properly.

For example, for textures, you may want to check the following

- Is the size appropriate?
- Are the compression settings appropriate?
- Are the MipMap settings appropriate?
- Read/Write settings are appropriate, etc.

Please refer to Chapter 4 "Tuning Practice - Asset" for more information on what to look out for in each asset.

1.6.2 GC (Mono)

If there is a lot of GC (Mono) in Simple View, it is likely that a large GC.Alloc is occurring at one time. Or memory may be fragmented due to GC.Alloc occurring every frame. These may be causing extra expansion of the managed heap area. In this case, you should steadily reduce GC.Alloc.

See "2.1.5 Memory" for more information on managed heap. Similarly, details on

GC.Alloc are handled at "2.5.1 Stack and Heap".

Notation Differences by Version

GC" is shown as "GC" in 2020.2 and later versions, while "Mono" is shown up to 2020.1 and below. Both refer to the amount of managed heap space occupied.

1.6.3 Other

Check for suspicious items in the Detailed View. For example, you may want to open the **Other** item once to investigate.

Name	Memory
▶ Assets (33256)	490.0 MB
▼ Other (792)	338.6 MB
System.ExecutableAndDlls	208.0 MB
▶ SerializedFile (592)	64.4 MB
▶ Profiling (8)	32.5 MB
▶ PersistentManager.Remapper (1)	16.0 MB
▶ Rendering (9)	8.2 MB
▶ Managers (28)	7.9 MB
Objects	1.3 MB
▶ MemoryPools (16)	105.4 KB
▶ Physics2D Module (1)	104.1 KB
▶ File System (2)	79.7 KB
▶ Job System (1)	2.9 KB
▶ ParticleSystem Module (2)	2.3 KB
▶ Animation Module (4)	1.7 KB
▶ MemoryProfiling (1)	96 B

▲ Figure 1.7 Other items

In my experience, SerializedFile and PersistentManager.Remapper were quite bloated. If you can compare values across multiple projects, it is a good idea to

1.7 Isolating the Cause of Processing Failures

do so once. Comparing the values of each may reveal outliers. See "2. Detailed view" for more information.

1.6.4 Plug-ins

So far we have used Unity's measurement tools to isolate the cause of the problem. However, Unity can only measure memory managed by Unity. In other words, the amount of memory allocated by plug-ins is not measured. Check to see if ThirdParty products are allocating extra memory.

Use the native measurement tool (Instruments in Xcode). See "3.7 Instruments" for more information.

1.6.5 Examine the specifications.

This is the last step. If there is nothing that can be cut from what we have covered so far, we have no choice but to consider the specifications. Here are some examples

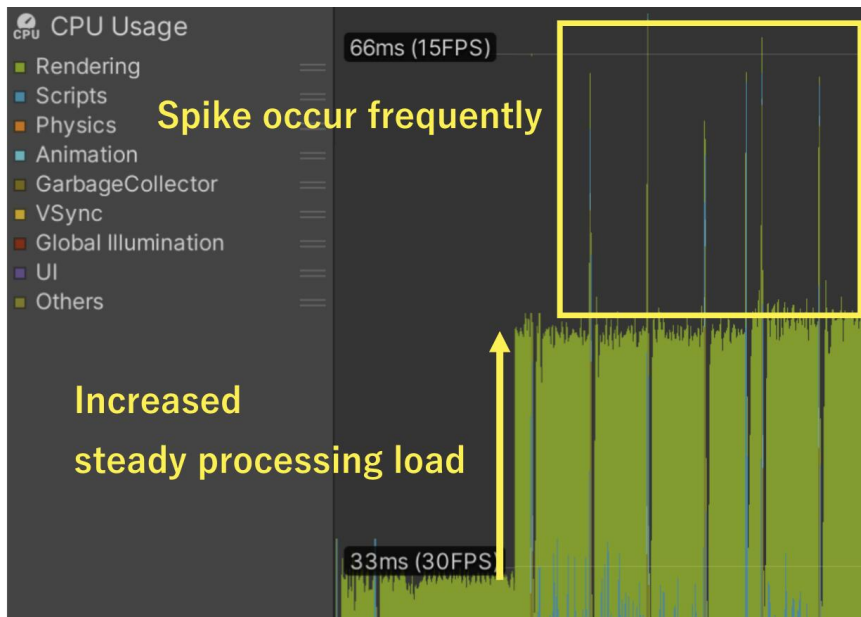
- Change the compression ratio of textures
 - Increase the compression ratio one step for one part of the texture
- Change the timing of loading/unloading
 - Release objects in resident memory and load them each time.
- Change load specifications
 - Reduce the number of character variations to be loaded ingame by one.

All of these changes have a large impact and may fundamentally affect the fun of the game. Therefore, specification considerations are a last resort. Make sure to estimate and measure memory early on to prevent this from happening.

1.7 Isolating the Cause of Processing Failures

The following is an introduction to the process of measuring and optimizing processing time. The way to deal with screen processing failures varies depending on whether they are "instantaneous" or "steady" processing failures.

Instantaneous processing slowdowns are measured as a processing load that is sharp like a needle. They are also called spikes because of their appearance.



▲ Figure 1.8 Spikes and steady processing load

Figure 1.8 In the following section, the measured data shows a sudden increase in steady-state load, as well as periodic spikes. Both events will require performance tuning. First, a relatively simple instantaneous load study will be explained. Then, the steady-state load survey will be explained.

1.8 Investigating Instantaneous Load

As a method of investigating spikes, Profiler (CPU) is used to investigate the cause.

See "3.1.2 CPU Usage" for detailed usage of the tool. First, isolate whether the cause is due to GC or not. Deep Profile is not necessary to isolate the cause itself, but it will be needed to solve the problem.

1.8.1 Spikes caused by GC

If GC (garbage collection) is occurring, GC.Alloc should be reduced. Deep Profile to see which processes are allocating how much. The first areas that should be reduced are those that are cost-effective. It is recommended to focus on the following items.

- Areas allocated every frame
- Areas where a large number of allocations are occurring

The fewer the allocations, the better, but this does not mean that allocations should be zero. For example, there is no way to prevent allocations that occur during the Instantiate process. In such cases, pooling, in which objects are used instead of generating objects each time, is effective. For more information on GC, please refer to "2.5.2 Garbage Collection".

1.8.2 Spikes due to heavy processing

If GC is not the cause, some kind of heavy processing is being performed instantaneously. Again, let's use Deep Profile to investigate what and how much processing is heavy, and review the areas that are taking the longest time to process.

The following are some of the most common temporary heavy processes.

- Instantiate processing
- Active switching of a large number of objects or objects in a deep hierarchy
- Screen capture processing, etc.

As this is a part that is highly dependent on the project code, there is no one-size-fits-all solution. If the actual measurement reveals the cause, please share the measurement results with the project members and discuss how to improve it.

1.9 Investigate the steady-state load

When improving the steady processing load, it is important to reduce the processing within a single frame. The processing performed within a single frame can be roughly divided into CPU processing and GPU processing. First, it is a good idea to isolate which of these two processes is the bottleneck, or which has the same processing load.

A situation where the CPU is the bottleneck is called CPU-bound, and a situation where the GPU is the bottleneck is called GPU-bound.

As an easy way to isolate the two, if any of the following apply to you, there is a good chance that you are GPU-bound.

- Dramatic improvement in processing load when the screen resolution is lowered
- When measured with Profiler **Gfx.WaitForPresent** is present

On the other hand, if these are not present, there is a possibility of CPU bounce. In the following sections, we will explain how to investigate CPU bounces and GPU bounces.

1.9.1 CPU Bound

CPU bound uses CPU (Profiler), which was also discussed in the previous section. It investigates using Deep Profile and checks whether a large processing load is applied to a specific algorithm. If there is no large processing load, it means that the system is equally heavy, so steadily improve the system. If you cannot reach the target reduction value even after making steady improvements, you may want to go back to "1.1.4 Determine quality setting specifications." and reconsider.

1.9.2 GPU Bound

In the case of GPU bounces, you may want to investigate using the Frame Debugger. See "3.3 Frame Debugger" for details on how to use it.

Is the resolution appropriate?

Among GPU bounces, resolution has a significant impact on the processing load of the GPU. Therefore, if the resolution is not set appropriately, the first priority should be to set it to an appropriate resolution.

First, check to see if the resolution is appropriate for the assumed quality settings. A good way to check is to look at the resolution of the render target being processed in the Frame Debugger. If you have not intentionally implemented the following, work

on optimization.

- Only UI elements are rendered at the full resolution of the device.
- Temporary textures for post-effects have a high resolution, etc.

Are there any unnecessary objects?

Check the Frame Debugger to see if there are any unnecessary drawings. For example, there may be an unneeded camera active that is drawing unrelated objects behind the scenes. If there are many cases where the previous drawing is wasted due to other obstructions, use the **Occlusion Culling** may be a good option. For more information on occlusion culling, see "7.5.3 Occlusion culling".

Note that occlusion culling requires data to be prepared in advance and that memory usage will increase as the data is deployed to memory. It is a common practice to build pre-prepared information in memory to improve performance in this way. Since memory and performance are often inversely proportional, it is a good idea to be aware of memory as well when employing something.

Is batching appropriate?

Batching is the process of drawing all the objects at once. Batching is effective for GPU bouncing because it improves drawing efficiency. For example, Static Batching can be used to combine the meshes of multiple immobile objects.

There are many different methods for batching, so I will list some of the most common ones. If you are interested in any of them, please refer to "7.3 Reducing Draw Calls".

- Static Batching
- Dynamic Batching
- GPU Instancing
- SRP Batcher, etc.

Look at the load individually

If the processing load is still high, the only way is to look at it individually. It may be that the object has too many vertices or that the shader processing is causing the problem. To isolate this, switch the active state of each object and see how the processing load changes. Specifically, we can try deactivating the background and see what happens, deactivate the characters and see what happens, and so on. Once the categories with high processing load are identified, the following factors should be further examined.

- Are there too many objects to draw?
 - Consider whether it is possible to draw them all at once.
- Is the number of vertices per object too large?
 - Consider reduction and LOD
- Is the processing load improved by replacing with a simple Shader?
 - Review the processing of Shader

Otherwise

It can be said that each GPU processing is piled up and heavy. In this case, the only way is to steadily improve one by one.

Also, as with CPU bound, if the target reduction cannot be reached, it is a good idea to go back to "1.1.4 Determine quality setting specifications." and reconsider.

1.10 Conclusion

In this chapter, we have discussed what to watch out for "before" and "during" performance tuning.

The things to watch out for before and during performance tuning are as follows

- Decide on "indicators," "guaranteed devices," and "quality setting specifications."
 - Verify and determine the indicators before mass production.
- Create a mechanism to easily notice performance degradation.

The following are things to keep in mind during performance tuning.

- Isolate the cause of performance degradation and take appropriate measures.
- Be sure to follow the sequence of "measurement," "improvement," and "re-measurement (checking the results)."

As explained up to this point, it is important to measure and isolate the cause of performance tuning. Even if a case not described in this document occurs, it will not be a major problem if the fundamentals are followed. If you have never done performance tuning before, we hope that you will put the information in this chapter into practice.



PERFORMANCE TUNING BIBLE

CHAPTER

02

第2章

Fundamentals

CyberAgent Smartphone Games & Entertainment

Chapter 2

Fundamentals

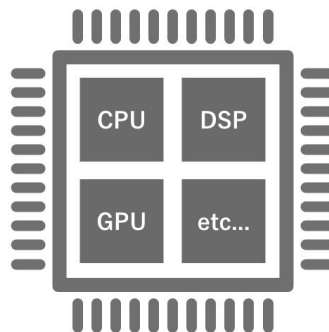
Performance tuning requires an examination and modification of the application as a whole. Therefore, effective performance tuning requires a wide range of knowledge, from hardware to 3D rendering to Unity mechanics. This chapter therefore summarizes the basic knowledge required to perform performance tuning.

2.1 Hardware

Computer hardware consists of five main devices: input devices, output devices, storage devices, computing devices, and control devices. These are called the five major devices of a computer. This section summarizes the basic knowledge of these hardware devices that are important for performance tuning.

2.1.1 SoC

A computer is composed of various devices. Typical devices include CPUs for control and computation, GPUs for graphics computation, and DSPs for processing audio and video digital data. In most desktop PCs and other devices, these are independent as separate integrated circuits, which are combined to form the computer. In smartphones, on the other hand, these devices are implemented on a single chip to reduce size and power consumption. This is called a system-on-a-chip, or SoC.



▲ Figure 2.1 SoC

2.1.2 iPhone, Android and SoC

The SoC used in a smartphone differs depending on the model.

For example, the iPhone uses a SoC called the A series designed by Apple. This series is named by combining the letter "A" and a number, such as A15, with the number getting larger as the version is upgraded.

In contrast, many Android devices use a SoC called Snapdragon. This SoC is manufactured by a company called Qualcomm and is named something like Snapdragon 8 Gen 1 or Snapdragon 888.

Also, while iPhones are manufactured by Apple, Android is manufactured by a variety of manufacturers. For this reason, Android has a variety of SoCs besides Snapdragon, as shown below Table 2.1. This is why Android is prone to model-dependent defects.

▼ Table 2.1 Major SoCs in Android

Series Name	Manufacturer	Trends in devices equipped with
Snapdragon	Qualcomm Inc.	Used in a wide range of devices
Helio	MediaTek	Used in some low-priced handsets
Kirin	HiSilicon	Huawei devices
Exynos	Samsung	Terminals by Samsung

When tuning performance, it is important to understand what is used in the device's SoC and what specifications it has.

The naming of Snapdragon has been a combination of the string "Snapdragon" and a three-digit number.

These numbers have a meaning: the 800s are the flagship models and are used in so-called high-end devices. The lower the number, the lower the performance and price, and the 400s are the so-called low-end handsets.

Even if a device is in the 400s, the performance improves with the newer release date, so it is difficult to make a general statement, but basically, the higher the number, the higher the performance.

Furthermore, it was announced in 2021 that the naming convention will be changed to something like Snapdragon 8 Gen 1 in the future, as this naming convention will soon run out of numbers.

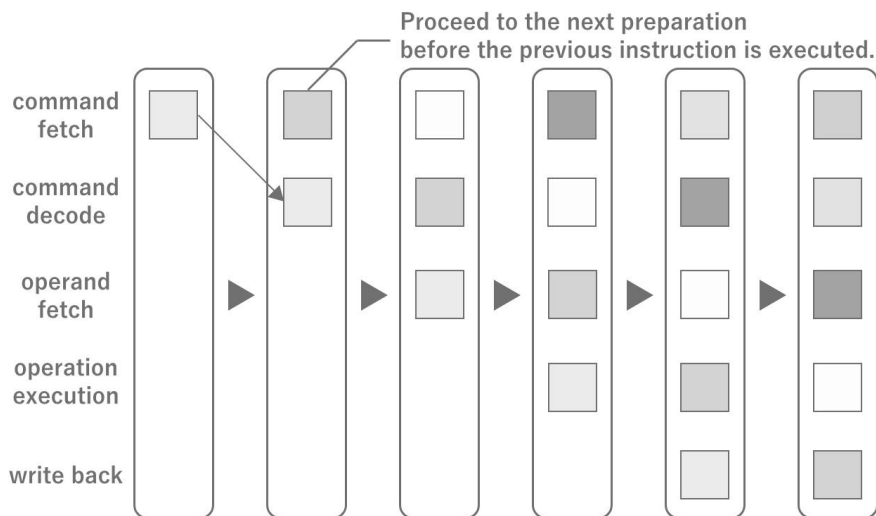
These naming conventions are useful to keep in mind when tuning performance, as they can be used as an indicator to determine the performance of a device.

2.1.3 CPU

CPU (Central Processing Unit) The CPU is the brain of the computer and is responsible not only for executing programs, but also for interfacing with the various hardware components of the computer. When actually tuning performance, it is useful to know what kind of processing is performed in the CPU and what kind of characteristics it has, so we will explain it from a performance perspective here.

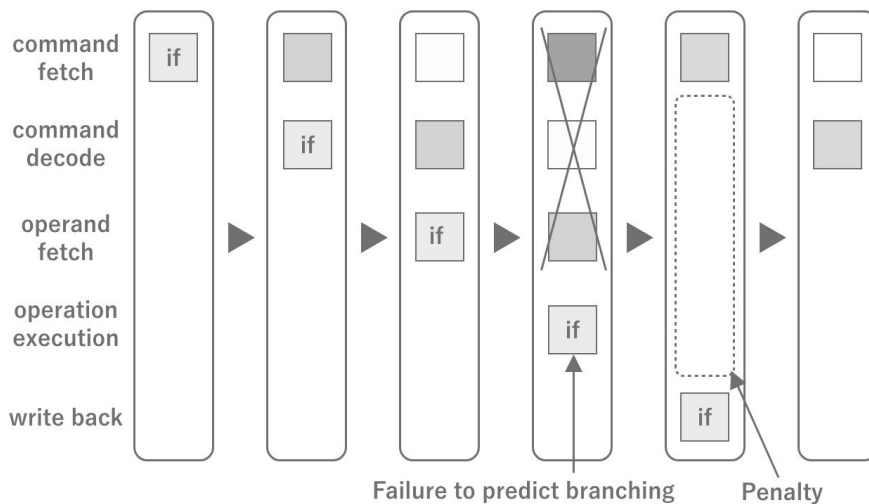
CPU Basics

What determines the execution speed of a program is not only simple arithmetic power, but also how fast it can execute the steps of a complex program. For example, there are four arithmetic operations in a program, but there are also branching operations. For the CPU, it does not know which instruction will be called next until it executes the program. Therefore, the hardware of the CPU is designed to be able to process a variety of instructions in rapid succession.



▲ Figure 2.2 CPU Pipeline Architecture

The flow of instructions inside the CPU is called a pipeline, and instructions are processed while predicting the next instruction in the pipeline. If the next instruction is not predicted, a pause called a pipeline stall occurs and the pipeline is reset. The majority of stalls are caused by branching. Although the branch itself anticipates the result to some extent, mistakes can still be made. Although performance tuning is possible without memorizing the internal structure, just knowing these things will help you to be more aware of how to avoid branching in loops when writing code.



▲ Figure 2.3 CPU Pipeline Stalling

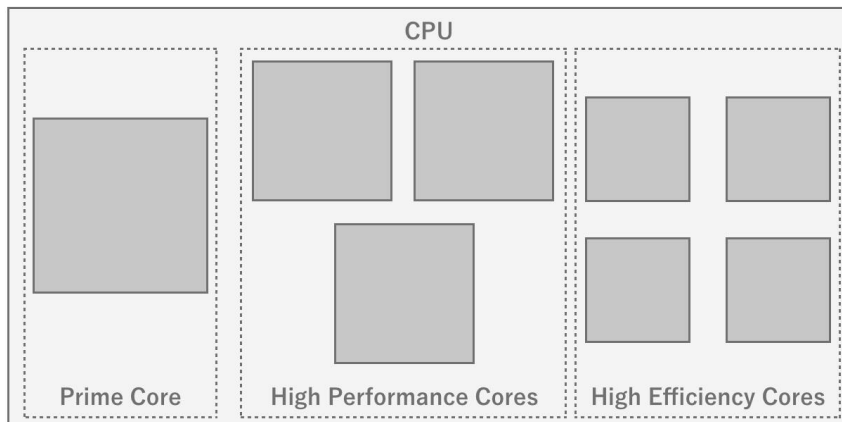
CPU computing power

The computing power of a CPU is determined by the clock frequency (unit: Hz) and the number of cores. The clock frequency indicates how many times per second the CPU can run. Therefore, the higher the clock frequency, the faster the program execution speed.

The number of cores, on the other hand, contributes to the parallel computing power of the CPU. A core is the basic unit in which a CPU operates, and when there is more than one it is called a multicore. Originally, there were only single cores, but in the case of single cores, in order to run multiple programs, the programs to be run alternately are switched. This is called a context switch, and its cost is very high. If you are used to smartphones, you may think that there is always one application (process) running, but in reality there are many different processes running in parallel, including the OS. Therefore, in order to provide optimal processing power even under such circumstances, multi-cores with multiple cores have become the mainstream. As of 2022, the mainstream for smartphones is around 2-8 cores.

In recent years, CPUs with asymmetric cores (big.LITTLE) have become the mainstream for multi-core processors (especially for smartphones). Asymmetric cores refer to CPUs that have a high-performance core and a power-saving core together.

The advantage of asymmetric cores is that normally only the power-saving cores are used to conserve battery power, and the cores can be switched when performance is required, such as in games. Note, however, that the maximum parallel performance is reduced by the power-saving cores, so the number of cores alone cannot be used to judge the performance of asymmetric cores.



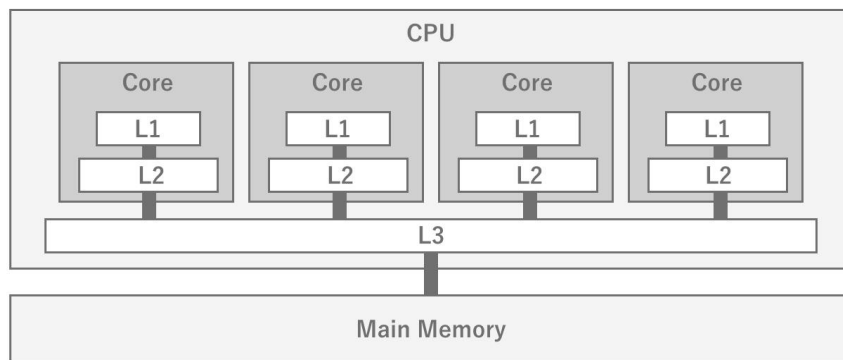
▲ Figure 2.4 Heterogeneous core configuration of Snapdragon 8 gen 1

Whether a program can use up multiple cores also depends on the parallel processing description of the program. For example, there are cases where the game engine has streamlined the physics engine by running it in a separate thread, or parallel processing is utilized through Unity's JobSystem, etc. Since the operation of the game's main loop itself cannot be parallelized, the higher performance of the core itself is advantageous even with multiple cores. Therefore, it is advantageous to have a high performance core itself, even if it is multi-core.

CPU Cache Memory

The CPU and main memory are physically located far apart and require a fraction of the time (latency) to access. Therefore, this distance becomes a major performance bottleneck when trying to access data stored in main memory during program execution. To solve this latency problem, a cache memory is installed inside the CPU. Cache memory mainly stores a portion of the data stored in main memory so that

programs can quickly access the data they need. There are three types of cache memory: L1, L2, and L3 cache. The smaller the number, the faster the speed, but the smaller the capacity. The smaller the number, the faster the cache, but the smaller the capacity. Therefore, the CPU cache cannot store all data, but only the most recently handled data.



▲ Figure 2.5 Relationship between the CPU L1, L2, and L3 caches and main memory

Therefore, the key to improving program performance is how to efficiently place data in the cache. Since the cache cannot be freely controlled by the program, data locality is important. In game engines, it is difficult to manage memory with an awareness of data locality, but some mechanisms, such as Unity's JobSystem, can achieve memory placement with enhanced data locality.

2.1.4 GPU

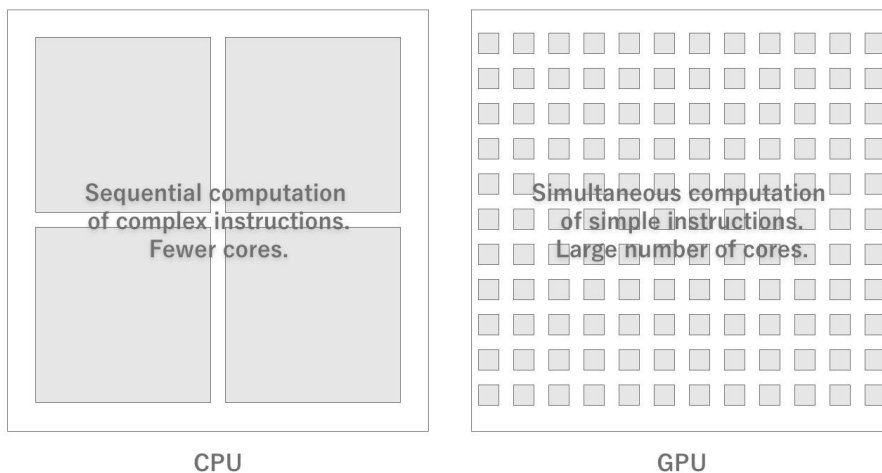
While CPUs specialize in executing programs **GPU** (Graphics Processing Unit) is a hardware specialized for image processing and graphics rendering.

GPU Basics

GPUs are designed to specialize in graphics processing, so their structure is very different from that of CPUs, and they are designed to process a large number of simple calculations in parallel. For example, if an image is to be converted to black and white, the CPU must read the RGB values of certain coordinates from memory, convert them to grayscale, and return them to memory, pixel by pixel. Since such a

process does not involve any branching and the calculation of each pixel does not depend on the results of other pixels, it is easy to perform the calculations for each pixel in parallel.

Therefore, GPUs can perform parallel processing that applies the same operation to a large amount of data at high speed, and as a result, graphics processing can be performed at high speed. In particular, graphics processing requires a large number of floating-point operations, and GPUs are particularly good at floating-point operations. For this reason, a performance index called FLOPS, which measures the number of floating-point operations per second, is generally used. Since it is difficult to understand the performance only in terms of computing power, an indicator called fill rate, which indicates how many pixels can be drawn per second, is also used.



▲ Figure 2.6 Difference between CPU and GPU

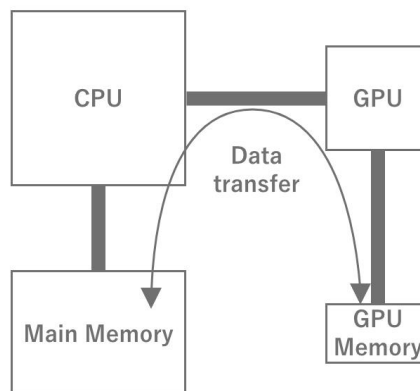
GPU Arithmetic Capacity

GPU hardware is characterized by a large number of cores (tens to thousands) that contain integer and floating-point arithmetic units. In order to deploy a large number of cores, the units required to run complex programs that were necessary for CPUs have been eliminated because they are no longer needed. Also, as with CPUs, the higher the clock frequency at which they operate, the more operations can be

performed per second.

GPU Memory

GPUs, of course, also require memory space for temporary storage to process data. Normally, this area is dedicated to the GPU, unlike main memory. Therefore, to perform any kind of processing, data must be transferred from main memory to GPU memory. After processing, the data is returned to main memory in the reverse order. Note that if the amount of data to be transferred is large, for example, transferring multiple high-resolution textures, the transfer takes time and becomes a processing bottleneck.



▲ Figure 2.7 GPU Memory Transfer

In mobile devices, however, the main memory is generally shared between the CPU and GPU, rather than being dedicated to the GPU. While this has the advantage of dynamically changing the memory capacity of the GPU, it has the disadvantage of sharing the transfer bandwidth between the CPU and GPU. In this case, data must still be transferred between the CPU and GPU memory areas.

GPGPU

GPUs can perform parallel operations on large amounts of data at high

speed, which CPUs are not good at. **GPGPU** (General Purpose GPU) GPGPU, General Purpose GPU. In particular, there are many cases where GPUs are used for machine learning such as AI and computational processing such as blockchain, which has led to a sharp increase in the demand for GPUs, resulting in a price hike and other effects. GPGPU can also be used in Unity by utilizing a function called Compute Shader.

2.1.5 Memory

Basically, all data is held in main memory, as the CPU only holds the data necessary for the calculation at that time. Since it is not possible to use more memory than the physical capacity, if too much is used, the memory cannot be allocated and the process is forced to terminate by the OS. This is generally referred to as **OOM** (Out Of Memory) This is commonly referred to as OOM, Out Of Memory, and is called "killed. As of 2022, the majority of smartphones are equipped with 4-8 GB of memory capacity. Even so, you should be careful not to use too much memory.

Also, as mentioned above, since memory is separated from the CPU, performance itself will vary depending on whether or not memory-aware implementation is used. In this section, we will explain the relationship between programs and memory so that performance-conscious implementation can be performed.

Memory Hardware

Although it is advantageous to have the main memory inside the SoC due to the physical distance, memory is not included in the SoC. There are reasons for this, such as the fact that the amount of memory installed cannot be changed from device to device if it is included in the SoC. However, if the main memory is slow, it will noticeably affect program execution speed, so a relatively fast bus is used to connect the SoC and memory. The memory and bus standards commonly used in smartphones are **LPDDR** is the LPDDR standard. There are several generations of LPDDR, but the theoretical transfer rate is several Gbps. Of course, it is not always possible to achieve the theoretical performance, but in game development, this is rarely a bottleneck, so there is no need to be aware of it.

Memory and OS

Within an OS, there are many processes running simultaneously, mainly system processes and user processes. The system processes play an important role in running the OS, and most of them reside in the OS as services and continue to run regardless of the user's intention. On the other hand, user processes are processes that are started by the user and are not essential for the OS to run.

There are two display states for apps on smartphones: foreground (foremost) and background (hidden). Generally, when a particular app is in the foreground, other apps are in the background. While an app is in the background, the process exists in a suspended state to facilitate the return process, and memory is maintained as it is. However, when the memory used by the entire system becomes insufficient, the process is killed according to the priority order determined by the OS. At this time, the most likely to be killed are user applications (\doteq games) in the background that are using a lot of memory. In other words, games that use a lot of memory are more likely to be killed when they are moved to the background, resulting in a worse user experience when returning to the game and having to start all over again.

If there is no other process to kill when it tries to allocate memory, it will be killed itself. In some cases, such as iOS, it is controlled so that no more than a certain percentage of the physical memory can be used by a single process. Therefore, there is a limit to the amount of memory that can be allocated. As of 2022, the limit for an iOS device with 3GB of RAM, which is a major RAM capacity, will be 1.3~1.4GB, so this is likely to be the upper limit for creating games.

Memory Swap

In reality, there are many different hardware devices, some of which have very small physical memory capacity. In order to run as many processes as possible on such terminals, the OS tries to secure virtual memory capacity in various ways. This is memory swap.

One method used in memory swap is memory compression. Physical capacity is saved by compressing and storing in memory, mainly memory that will not be accessed for a while. However, because of the compression and decompression costs, it is not done for areas that are actively used, but for applications that have gone to the background, for example.

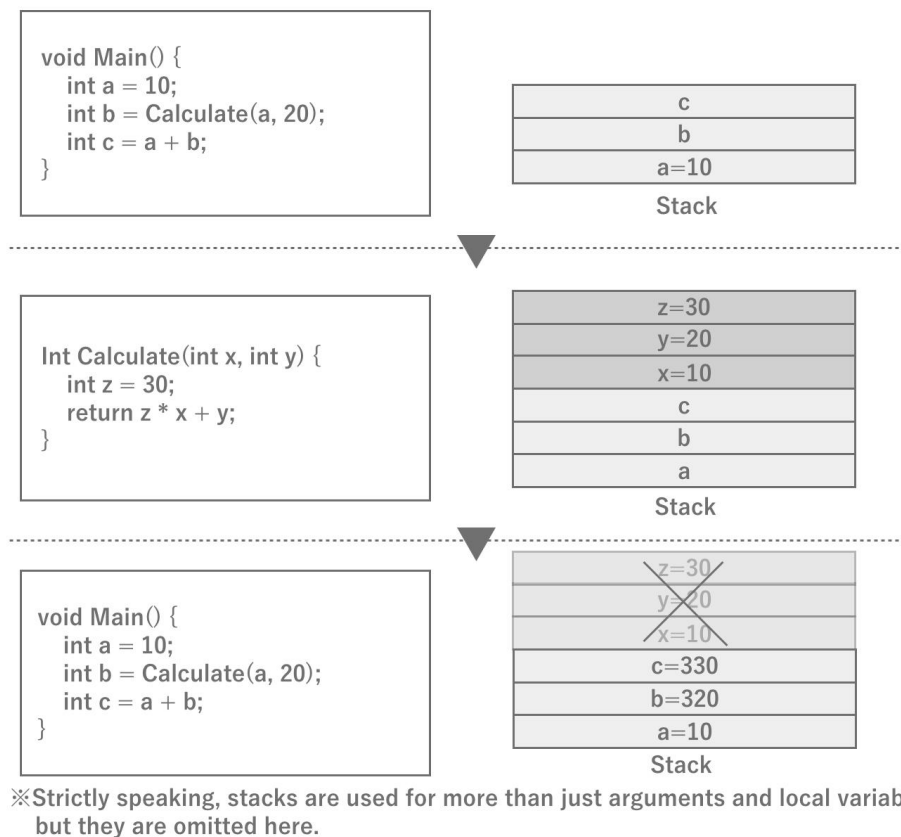
Another technique is to save storage of unused memory. On hardware with ample

Chapter 2 Fundamentals

storage, such as a PC, instead of terminating processes to free up memory, it may try to free up physical memory by saving unused memory to storage. This has the advantage of securing a larger amount of memory than memory compression, but it is not used because storage is slower than memory, so there are strong performance limitations, and it is not very realistic for smartphones, which have a small storage size to begin with.

Stack and Heap

Stack and **heap** you may have heard the terms "stack" and "heap" at least once. The stack is actually a dedicated fixed area that is closely related to the operation of the program. When a function is called, the stack is allocated for arguments and local variables, and when the function returns to the original function, the stack is released and the return value is accumulated. In other words, when the next function is called within a function, the information of the current function is left as it is and the next function is loaded into memory. In this way, the function call mechanism is realized. Stack memory depends on the architecture, but since the capacity itself is very small (1 MB), only a limited amount of data is stored.

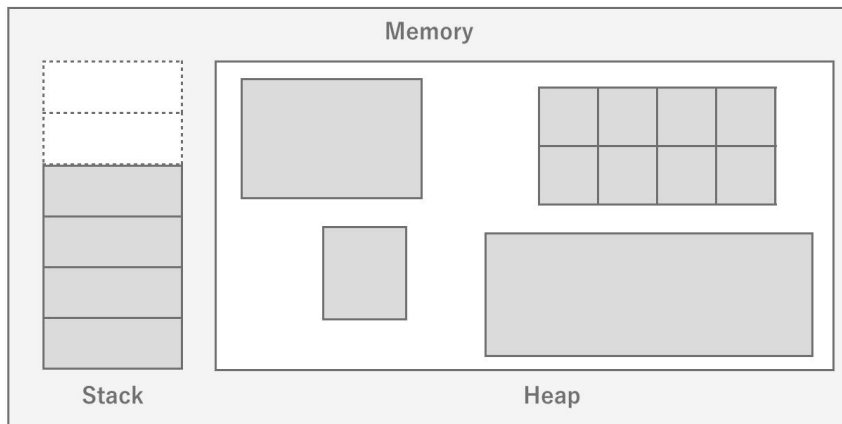


▲ Figure 2.8 Schematic diagram of stack operation

The heap, on the other hand, is a memory area that can be freely used within the program. Whenever the program needs it, it can issue a memory allocation instruction (malloc in C) to allocate and use a large amount of data. Of course, when the program finishes using the memory, it needs to release it (free). In C#, memory allocation and deallocation are automatically performed at runtime, so implementors do not need to do this explicitly.

Since the OS does not know when and how much memory is needed, it allocates memory from the free space when it is needed. If the memory cannot be allocated continuously when the memory allocation is attempted, it is assumed to be out of memory. This keyword "consecutive" is important. In general, repeated allocation and deallocation of memory results in **memory fragmentation** occurs. When mem-

ory is fragmented, even if the total free space is sufficient, there may be no contiguous free space. In such a case, the OS will first try to **Heap expansion** to the heap. In other words, it allocates new memory to be allocated to processes, thereby ensuring contiguous space. However, due to the finite memory of the entire system, the OS will kill the process if there is no more memory left to allocate.



▲ Figure 2.9 Stack and Heap

There is a noticeable difference in memory allocation performance when comparing stack and heap. This is because the amount of stack memory required for a function is determined at compile time, so the memory area is already allocated, whereas the heap does not know the amount of memory required until execution, so the heap allocates memory by searching for it in the free area each time. This is why heap is slow and stack is fast.

Stack Overflow Error

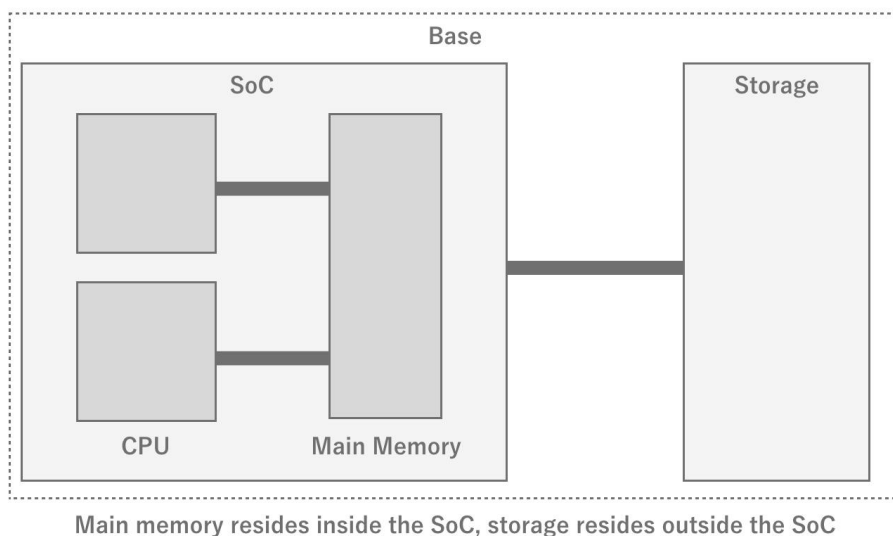
The Stack Overflow error occurs when stack memory is used up due to recursive calls to functions. The default stack size for iOS/Android is 1MB, so this error is more likely to occur when the size of recursive calls increases. In general, it is possible to prevent this error by changing to an algorithm that

does not result in recursive calls, or by changing to an algorithm that does not allow recursive calls to become too deep.

2.1.6 Storage

When you actually proceed with tuning, you may notice that it often takes a long time to read a file. Reading a file means reading data from the storage where the file is stored and writing it to memory so that it can be handled by the program. Knowing what is actually happening there is useful when tuning.

First, a typical hardware architecture will have dedicated storage for persistent data. Storage is characterized by its large capacity and its ability to persist data without a power supply (nonvolatile). Taking advantage of this feature, a vast amount of assets as well as the program of the application itself are stored in the storage, and are loaded from the storage and executed at startup, for example.



▲ Figure 2.10 Relationship between SoC and Storage

RAM and ROM

Especially in Japan, it is common to write "RAM" for smartphone memory and "ROM" for storage, but ROM actually refers to Read Only Memory. As the name suggests, it is supposed to be read-only and not writable, but the use of this term seems to be strongly customary in Japan.

However, the process of reading and writing to this storage is very slow compared to the program execution cycle from several perspectives.

- The physical distance from the CPU is greater than that of memory, resulting in large latency and slow read/write speeds.
- There is a lot of waste because reads are done in block units, including the commanded data and its surroundings.
- Sequential read/write is fast, while random read/write is slow.

The fact that random read/write is slow is particularly important. To begin with, sequential read/write and random read/write are sequential when a single file is read/written in order from the beginning of the file. However, when reading/writing multiple parts of a single file or reading/writing multiple small files at once, it is random. If you are reading/writing multiple parts of a file, or reading/writing multiple small files, it will be random. It is important to note that even when reading/writing multiple files in the same directory, they may not be physically located consecutively, so if they are physically far apart, they will be randomized.

The process of reading from storage

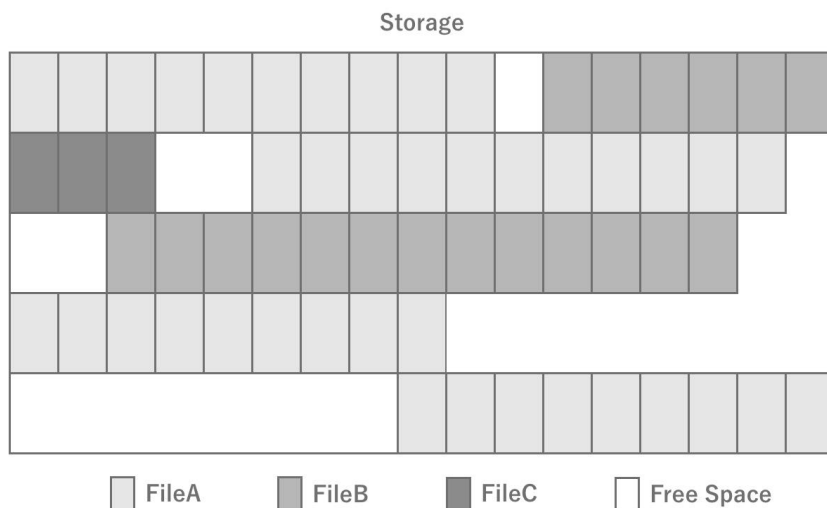
When reading a file from storage, the details are omitted, but the process roughly follows the flow below.

The 1. program commands the storage controller the area of the file to be read from storage The 2. storage controller receives the command and calculates the area to be read on the physical where the data is located 3. Reads the data 4. Writes the data in memory The 5. program accesses the

data through memory

There may also be more layers, such as controllers, depending on the hardware and architecture. It is not necessary to remember them exactly, but be aware that there are more hardware processing steps compared to reading from memory.

Also, typical storage achieves performance and space efficiency by writing a single file in blocks of 4KB or so. These blocks are not necessarily physically contiguous, even if they are a single file. The state in which files are physically distributed is called **fragmentation** (fragmentation) and the operation to eliminate fragmentation is called **defragmentation**. While fragmentation was often a problem with HDDs, which were the mainstay of PCs, it has almost disappeared with the advent of flash storage. While we do not need to be aware of file fragmentation in smartphones, it is important to be aware of it when considering PCs.



▲ Figure 2.11 Storage fragmentation

Types of storage in PCs and smartphones

In the PC world, HDDs and SSDs are the most common types of storage; you may not have seen HDDs before, but they are media that are recorded in the form of disks, like CDs, with heads that move over the disks to read the magnetism. As such, it was a device that was structurally large and had high latency due to the physical movement involved. In recent years, SSDs have become popular. Unlike HDDs, SSDs do not generate physical movement and therefore offer high-speed performance, but on the other hand, they have a limit to the number of read/write cycles (lifespan), so they become unusable when frequent read/write cycles occur. Although smartphones are different from SSDs, they use a type of flash memory called NAND.

Finally, how fast is the actual read/write speed of storage in a smartphone? As of 2022, one estimate is about 100 MB/s for reading. If you want to read a 10 MB file, it will take 100 ms to read the entire file, even under ideal conditions. Furthermore, if multiple small files are to be read, random accesses will occur, making the reading process even slower. Thus, it is always good to be aware that it actually takes a surprisingly long time to read a file. As for the specific performance of individual terminals, you can refer to ^{*1}, a site that collects benchmark results.

Finally, to summarize, when reading and writing files, it is a good idea to be aware of the following points

- Storage read/write speeds are surprisingly slow, and do not expect the same speed as memory
- Reduce the number of files to be read/written at the same time as much as possible (e.g., distribute timing, consolidate files into a single file, etc.)

2.2 Rendering

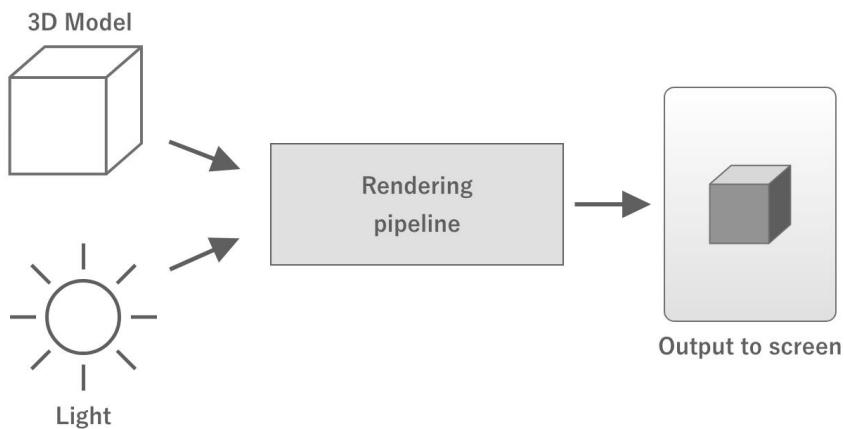
In games, rendering workloads often have a negative impact on performance. Therefore, knowledge of rendering is essential for performance tuning. Therefore, this

^{*1} https://maxim-saplin.github.io/cpdt_results/

section summarizes the fundamentals of rendering.

2.2.1 Rendering Pipeline

In computer graphics, a series of processes are performed on data such as the vertex coordinates of a 3D model and the coordinates and colors of lights to finally output the colors to be output to each pixel on the screen. This processing mechanism is called the **rendering pipeline** is called the rendering pipeline.



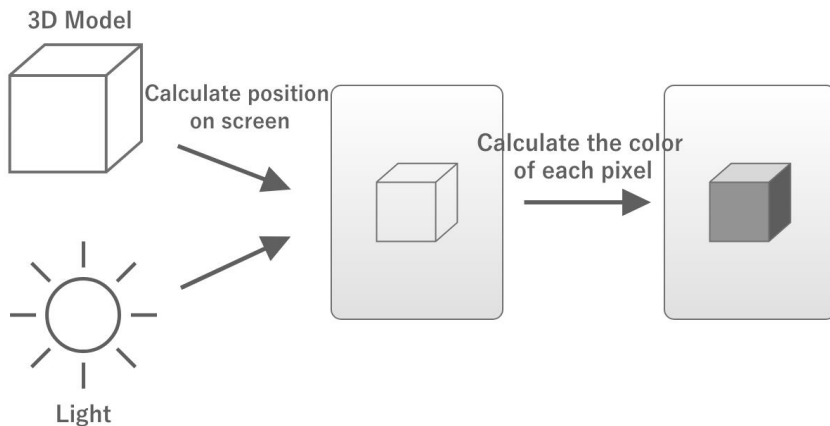
▲ Figure 2.12 Rendering Pipeline

The rendering pipeline starts with sending the necessary data from the CPU to the GPU. This data includes the coordinates of the vertices of the 3D model to be rendered, the coordinates of the lights, the material information of the objects, the camera information, and so on.

At this point, the data sent are the coordinates of the 3D model's vertices, camera coordinates, orientation, angle of view, etc., each of which is individual data. The GPU compiles this information and calculates where the object will appear on the screen when it is viewed with the camera. This process is called coordinate transformation.

Once the position of the object on the screen is determined, the next step is to determine the color of the object. The GPU then calculates the color of the object by asking, "What color will the corresponding pixels on the screen be when the light

illuminates the model?



▲ Figure 2.13 Calculating Position and Color

In the above process, "where on the screen the object will appear" is determined by the **Vertex Shader** and "the color of the area corresponding to each pixel on the screen" is calculated by a program called **fragment shader** and "what color the corresponding part of each pixel on the screen will be" is calculated by a program called the fragment shader.

These shaders can be freely written. Therefore, writing heavy processing in the vertex shaders and fragment shaders will increase the processing load.

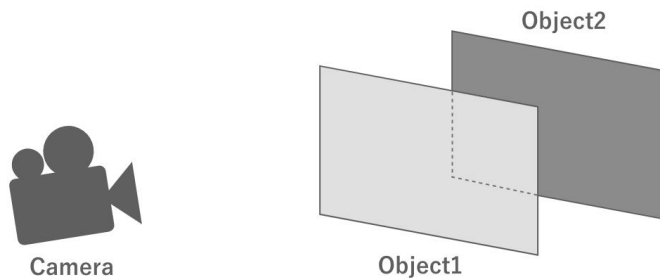
Also, the vertex shader processes the number of vertices in the 3D model, so the more vertices there are, the greater the processing load. Fragment shaders increase the processing load as the number of pixels to be rendered increases.

Actual rendering pipeline

In the actual rendering pipeline, there are many processes other than vertex shaders and fragment shaders, but since the purpose of this document is to understand the concepts necessary for performance tuning, we will only give a brief description.

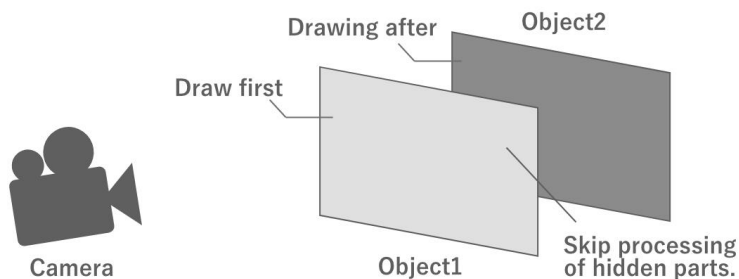
2.2.2 Semi-transparent rendering and overdraw

When rendering, the transparency of the object in question is an important issue. For example, consider two objects that are partially overlapped when viewed from the camera.



▲ Figure 2.14 Two overlapping objects

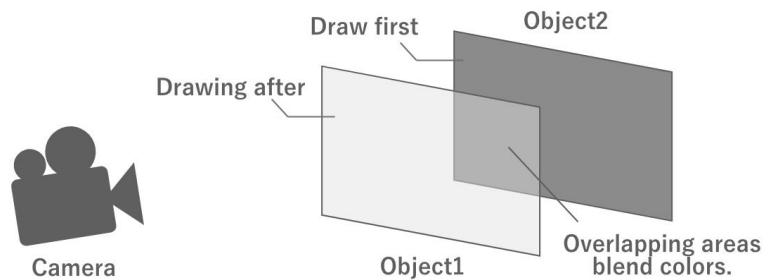
First, consider the case where both of these objects are opaque. In this case, the objects in front of the camera are drawn first. In this way, when drawing the object in the back, the part of the object that is not visible because it overlaps the object in the front does not need to be processed. This means that the fragment shader operation can be skipped in this area, thus optimizing the processing load.



▲ Figure 2.15 Opaque rendering

On the other hand, if both objects are semi-transparent, it would be unnatural if the back object is not visible through the front object, even if it overlaps the front object.

In this case, the drawing process is performed starting with the object in the back as seen from the camera, and the color of the overlapping area is blended with the already drawn color.



▲ Figure 2.16 Semi-transparent rendering

Unlike opaque rendering, semi-transparent rendering requires rendering of overlapping objects. If there are two semi-transparent objects that fill the entire screen, the entire screen will be processed twice. Thus, drawing semi-transparent objects on top of each other is called **overdraw** is called "overdraw. Too many overdraws can put a heavy processing load on the GPU and lead to performance degradation, so it is necessary to set appropriate regulations when drawing semi-transparent objects.

Assuming forward rendering

There are several ways to implement the rendering pipeline. Of these, the description in this section assumes forward rendering. Some points may not be partially applicable to other rendering methods such as deferred rendering.

2.2.3 Draw calls, set-pass calls, and batching

Rendering requires a processing load not only on the GPU but also on the CPU.

As mentioned above, when rendering an object, the CPU sends commands to the GPU to draw. This is called a **draw call** and is executed as many times as the number of objects to be rendered. At this time, if the texture or other information is

different from that of the object rendered in the previous draw call, the GPU will set the texture or other information to the GPU. This is done using the **set path call** and is a relatively heavy process. Since this process is done on the CPU's render thread, it is a processing load on the CPU, and too much of it can affect performance.

Unity has a feature to reduce draw calls called **draw call batching** to reduce draw calls. This is a mechanism whereby meshes of objects with the same texture and other information (i.e., the same material) are combined in CPU-side processing in advance and drawn with a single draw call. Batching at runtime **Dynamic batching** and the merged mesh is created in advance. **Static batching** which creates a combined mesh in advance.

There is also a **Scriptable Render Pipeline** also has an **SRP Batcher** mechanism. Using this mechanism, set-pass calls can be combined into a single call, even if the mesh and material are different, as long as the shader variants are the same. This mechanism does not reduce the number of draw calls, but it does reduce the set-pass calls, since these are the ones that are the most expensive to process.

For more information on these batching arrangements, see "7.3 Reducing Draw Calls".

GPU Instancing

A feature that has a similar effect to batching is **GPU Instancing** is GPU instancing. This function uses the GPU's ability to draw objects with the same mesh in a single draw call or set-path call.

2.3 Data Representation

Games use a variety of data, including images, 3D models, audio, and animation. Knowing how these are represented as digital data is important for calculating memory and storage capacity and for properly configuring settings such as compression. This section summarizes the basic data representation methods.

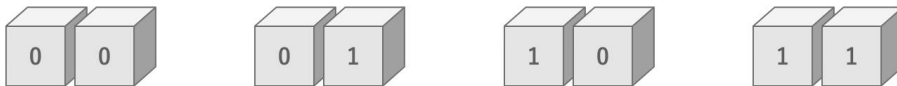
2.3.1 Bits and Bytes

The smallest unit a computer can represent is the bit. A bit can represent the range of data that can be represented by a single binary digit, i.e., a combination of 0 and 1. This can only represent simple information, such as turning a switch on and off, for example.



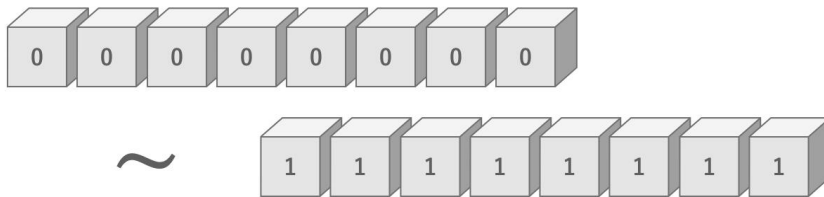
▲ Figure 2.17 Amount of information in one bit

If we use two bits, we can express the range that can be represented by two digits of binary numbers, in other words, four combinations. Since there are four combinations, it is possible to express, for example, which key was pressed: up, down, left, or right.



▲ Figure 2.18 2 bits of information

Similarly, 8 bits can represent a range of 8 binary digits, i.e., 2^8 digits = 256 ways. At this point, it seems that a variety of information can be expressed. These 8 bits are expressed in the unit of 1 byte. In other words, one byte is a unit that can express 256 different amounts of information.



▲ Figure 2.19 Amount of information in 8 bits

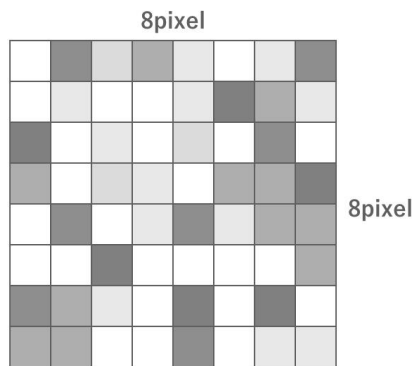
There are also units that represent larger numbers, such as the kilobyte (KB), which represents 1,000 bytes, and the megabyte (MB), which represents 1,000 kilobytes.

Kilobyte and Kibibyte

Above, 1 KB is written as 1,000 bytes, but in some contexts, 1 KB may be referred to as 1,024 bytes. When explicitly referring to them differently, 1,000 bytes is called 1 kilobyte (KB) and 1,024 bytes is called 1 kibibyte (KiB). The same is true for megabytes.

2.3.2 Image

Image data is represented as a set of pixels. For example, an 8×8 pixel image consists of a total of $8 \times 8 = 64$ pixels.

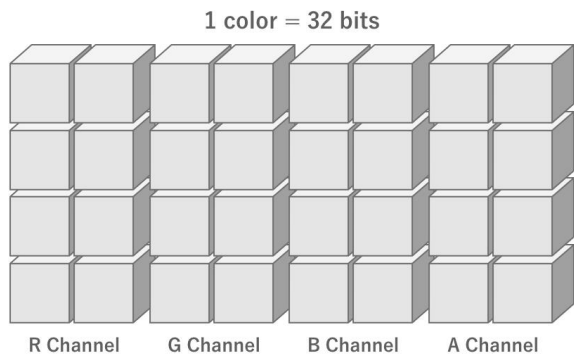


▲ Figure 2.20 Image Data

In this case, each pixel has its own color data. So how is color represented in digital data?

First, color is created by combining four elements: red (Red), green (Green), blue (Blue), and transparency (Alpha). These are called channels, and each channel is represented by the initial letters RGBA.

In the commonly used True Color method of color representation, each RGBA value is expressed in 256 steps. As explained in the previous section, 256 steps means 8 bits. In other words, True Color can be represented with $4 \text{ channels} \times 8 \text{ bits} = 32 \text{ bits}$ of information.



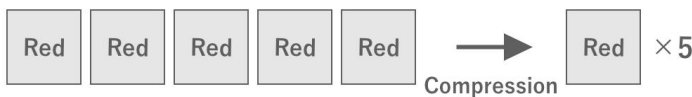
▲ Figure 2.21 Amount of information per color

Thus, for example, an 8×8 pixel True Color image has $8 \text{ pixels} \times 8 \text{ pixels} \times 4 \text{ channels} \times 8 \text{ bits} = 2,048 \text{ bits} = 256 \text{ bytes}$. For a $1,024 \times 1,024$ pixel True Color image, its information content would be $1,024 \text{ pixels} \times 1,024 \text{ pixels} \times 4 \text{ channels} \times 8 \text{ bits} = 33,554,432 \text{ bits} = 4,194,304 \text{ bytes} = 4,096 \text{ kilobytes} = 4 \text{ megabytes}$.

2.3.3 Image Compression

In practice, images are most often used as compressed data.

Compression is the process of reducing the amount of data by devising a way to store the data. For example, suppose there are five pixels with the same color next to each other. In this case, rather than having five color information for each pixel, it is better to have one color information and the information that there are five pixels in a row, which reduces the amount of information.



▲ Figure 2.22 Compression

In reality, there are many more complex compression methods.

As a concrete example, let us introduce ASTC, a typical mobile compression format. Applying the format ASTC6x6, a 1024×1024 texture is compressed from 4 megabytes to about 0.46 megabytes. In other words, the result is that the capacity is compressed to less than one-eighth of its original size, and we can recognize the importance of compression.

For reference, the compression ratio of the ASTC format, which is mainly used in mobile devices, is described below.

▼ Table 2.2 Compression Format and Compression Ratio

Compression Format	Compression ratio
ASTC RGB(A) 4x4	0.25
ASTC RGB(A) 6x6	0.1113
ASTC RGB(A) 8x8	0.0625
ASTC RGB(A) 10x10	0.04
ASTC RGB(A) 12x12	0.0278

In Unity, various compression methods can be specified for each platform using the texture import settings. Therefore, it is common to import an uncompressed image and apply compression according to the import settings to generate the final texture to be used.

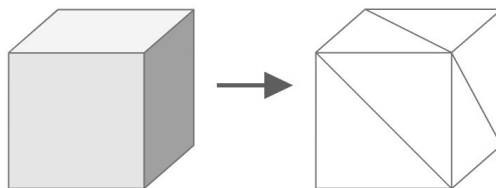
GPUs and Compression Formats

Images compressed according to a certain rule must, of course, be decompressed according to that rule. This decompression is done at runtime. To minimize this processing load, it is important to use a compression format that is supported by the GPU. ASTC is a typical compression format supported by GPUs on mobile devices.

2.3.4 Mesh

In 3DCG, a three-dimensional shape is expressed by connecting many triangles in 3D space. This collection of triangles is called a **mesh**.

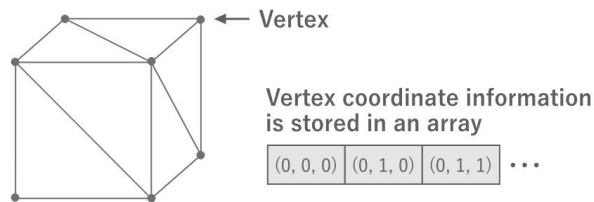
A three-dimensional object is composed of triangles



▲ Figure 2.23 3D by combining triangles

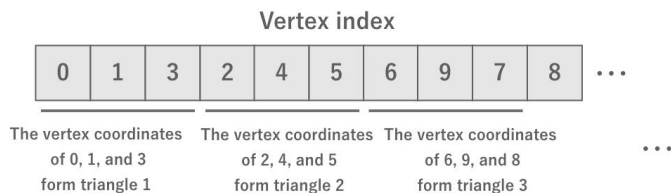
The triangles can be represented as the coordinate information of three points in 3D space. Each of these points is called a **vertex** and its coordinates are called **vertex coordinates** and their coordinates are called vertex coordinates. All vertex information per mesh is stored in a single array.

2.3 Data Representation



▲ Figure 2.24 Vertex Information

Since the vertex information is stored in a single array, we need additional information to indicate which of the vertices will be combined to form a triangle. This is called the **vertex index** and is represented as an array of type `int` that represents the index of the array of vertex information.



▲ Figure 2.25 Vertex Index

Additional information is needed for texturing and lighting objects. For example, mapping a texture requires UV coordinates. Lighting also requires information such as vertex color, normals, and tangents.

The following table summarizes the main vertex information and the amount of information per vertex.

▼ Table 2.3 Vertex Information

Name	Amount of information per vertex
Vertex coordinates	3D float = 12 bytes
UV coordinates	2D float = 8 bytes
Vertex color	4-dimensional float = 16 bytes
Normal	3-dimensional float = 12 bytes
Tangent	3D float = 12 bytes

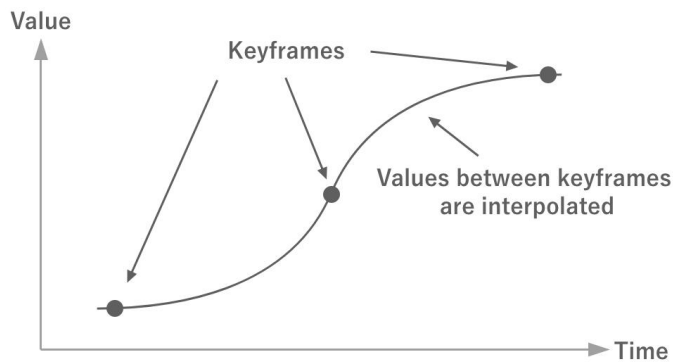
It is important to determine the number of vertices and the type of vertex informa-

tion in advance because mesh data grows as the number of vertices and the amount of information handled by a single vertex increases.

2.3.5 Keyframe Animation

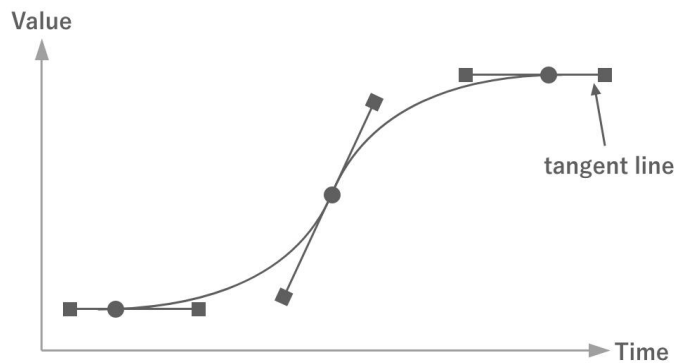
Games use animation in many areas, such as UI animation and 3D model motion. Keyframe animation is one of the most common ways to achieve animation.

A keyframe animation consists of an array of data representing values at a certain time (keyframe). The values between keyframes are obtained by interpolation and can be treated as if they were smooth, continuous data.



▲ Figure 2.26 Keyframes

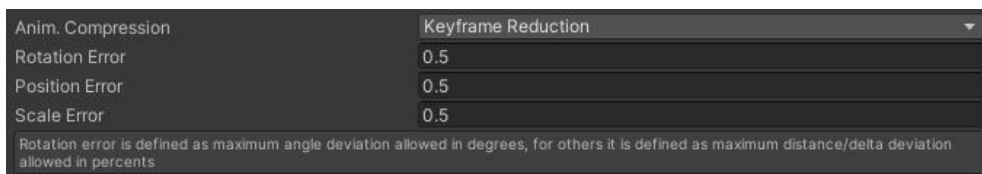
In addition to time and value, keyframes have other information such as tangents and their weights. By using these in the interpolation calculation, more complex animations can be realized with less data.



▲ Figure 2.27 Tangents and Weights

In keyframe animation, the more keyframes there are, the more complex the animation can be. However, the amount of data also increases with the number of keyframes. For this reason, the number of keyframes should be set appropriately.

There are methods to compress the amount of data by reducing the number of keyframes while keeping the curves as similar as possible. In Unity, keyframes can be reduced in the model import settings as shown in the following figure.



▲ Figure 2.28 Import Settings

See "4.4 Animation" for details on how to set up the settings.

2.4 How Unity Works

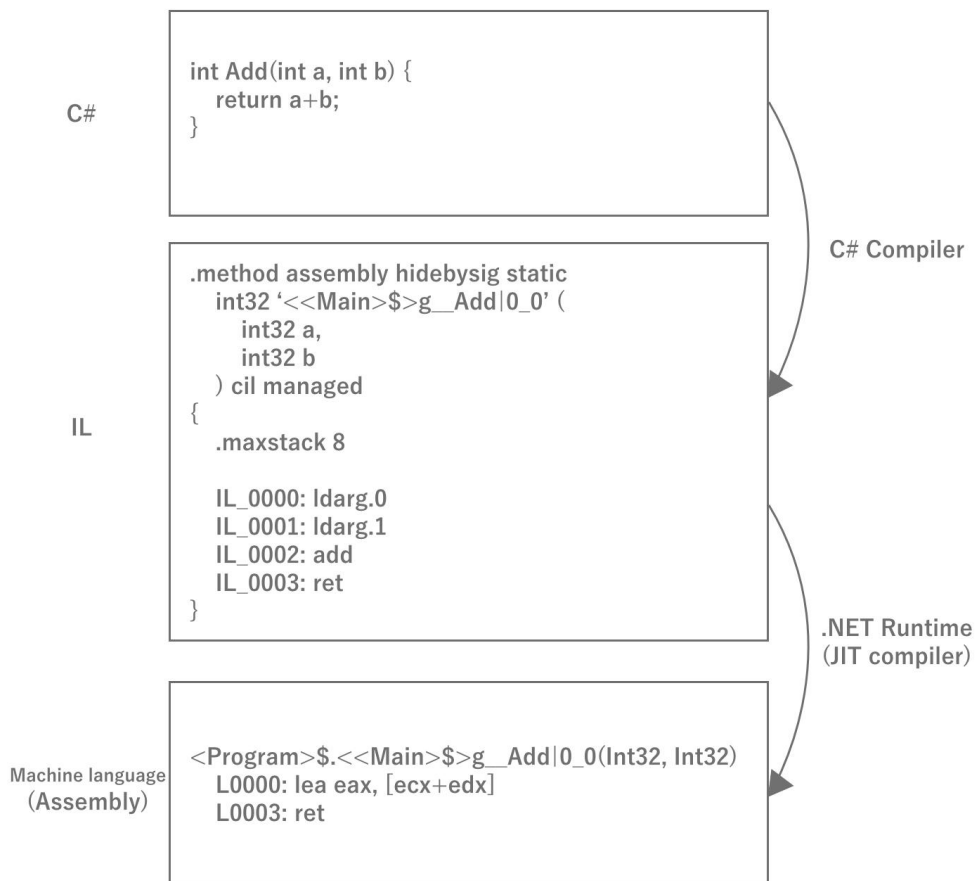
Understanding how the Unity engine actually works is obviously important for tuning your game. This section explains the principles of Unity's operation that you should know.

2.4.1 Binaries and Runtime

First of all, this section explains how Unity actually works and how the runtime works.

C# and the Runtime

When creating a game in Unity, developers program the behavior in C#. C# is a compiler language, as it is often compiled (built) when developing games in Unity. However, C# differs from traditional C and other languages in that it is not a machine language that can be compiled and executed by itself on a machine, but rather it is an **. Intermediate Language; henceforth IL**. The executable code that is converted to IL is called the "executable code". Since the executable code converted to IL cannot be executed by itself, it is executed while being converted to machine language using the **. runtime**.



▲ Figure 2.29 C# Compilation Process

The reason for interrupting IL once is that once converted to machine language, the binary can only be executed on a single platform. With IL, any platform can run simply by preparing a runtime for that platform, eliminating the need to prepare binaries for each platform. Therefore, the basic principle of Unity is that IL obtained by compiling the source code is executed on the runtime for the respective environment, thereby achieving multi-platform compatibility.

Let's check the IL code.

IL code, which is usually rarely seen, is very important to be aware of performance such as memory allocation and execution speed. For example, an array and a list will output different IL code for the same foreach loop at first glance, with the array being the better performing code. You may also find unintended hidden heap allocations. In order to acquire a sense of the correspondence between C# and IL code, it is recommended to check the IL conversion results of C# code you have written on a regular basis. You can view IL code in IDEs such as Visual Studio or Rider, but IL code itself is a difficult language to understand because it is a low-level language called assembly. In such cases, you can use a web service called SharpLab^{*2} to check C# -> IL -> C# and vice versa to make it easier to understand the IL code. An actual example of the conversion is presented at Chapter 10 "Tuning Practice - Script (C#)" in the latter half of this document.

IL2CPP

As mentioned above, Unity basically compiles C# into IL code and runs it at runtime, but starting around 2015, some environments started having problems. That is 64-bit support for apps running on iOS and Android. As mentioned above, C# requires a runtime to run in each environment to execute IL code. In fact, until then, Unity was actually a long-standing OSS implementation of the . **Mono** NET Framework OSS implementation, and Unity itself modified it for its own use. In other words, in order for Unity to become 64-bit compatible, it was necessary to make the forked Mono 64-bit compatible. Of course, this would require a tremendous amount of work, so Unity decided to use **IL2CPP**. Unity overcame this challenge by developing a technology called IL2CPP instead.

IL2CPP is, as the name suggests, IL to CPP, a technology that converts IL code to C++ code. Since C++ is a highly versatile language that is natively supported in any development environment, it can be compiled into machine language in each development tool chain once it is output to C++ code. Therefore, 64-bit support is

^{*2} <https://sharplab.io/>

the job of the toolchain, and Unity does not have to deal with it. Unlike C#, C++ code is compiled into machine language at build time, eliminating the need to convert it to machine language at runtime and improving performance.

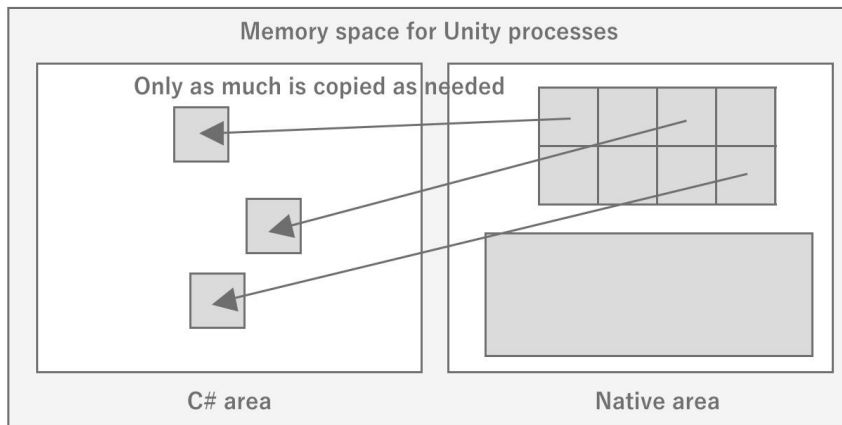
Although C++ code generally has the disadvantage of taking a long time to build, the IL2CPP technology has become a cornerstone of Unity, solving 64-bit compatibility and performance in one fell swoop.

Unity Runtime

By the way, although Unity allows developers to program games in C#, the runtime of Unity itself, called the engine, does not actually run in C#. The source itself is written in C++, and the part called the player is distributed pre-built to run in each environment. There are several possible reasons why Unity writes its engine in C++.

- To get fast and memory-saving performance
- To support as many platforms as possible
- To protect the intellectual property rights of the engine (black box)

Since the C# code written by the developer runs in C#, Unity requires two areas: the engine part, which runs natively, and the user code part, which runs at C# runtime. The engine and user code work by exchanging data as needed during execution. For example, when `GameObject.transform` is called from C#, all game execution state such as scene state is managed inside the engine, so first makes a native call to access memory data in the native area and then returns values to C#. It is important to note that memory is not shared between C# and native, so data needed by C# is allocated on the C# side each time it is needed. API calls are also expensive, with native calls occurring, so an optimization technique of caching values without frequent calls is necessary.



▲ Figure 2.30 Image of memory state in Unity

In this way, when developing Unity, it is necessary to be aware of the invisible engine part to some extent. For this reason, it is a good idea to look at the source code of the interface between the native Unity engine and C#. Fortunately, Unity has made the C# part of the source code available on GitHub at ^{*3}, so you can see that it is mostly native calls, which is very helpful. I recommend making use of this if necessary.

2.4.2 Asset entities

As explained in the previous section, since the Unity engine runs natively, it basically has no data on the C# side. The same is true for the handling of assets: assets are loaded in the native area, and only references are returned to C#, or data is copied and returned. Therefore, there are two main ways to load assets: by specifying a path to load them on the Unity engine side, or by passing raw data such as byte arrays directly to the engine. If a path is specified, the C# side does not consume memory because it is loaded in the native area. However, if data such as a byte array is loaded and processed from the C# side and passed to the C# side, memory is doubly consumed on both the C# and native sides.

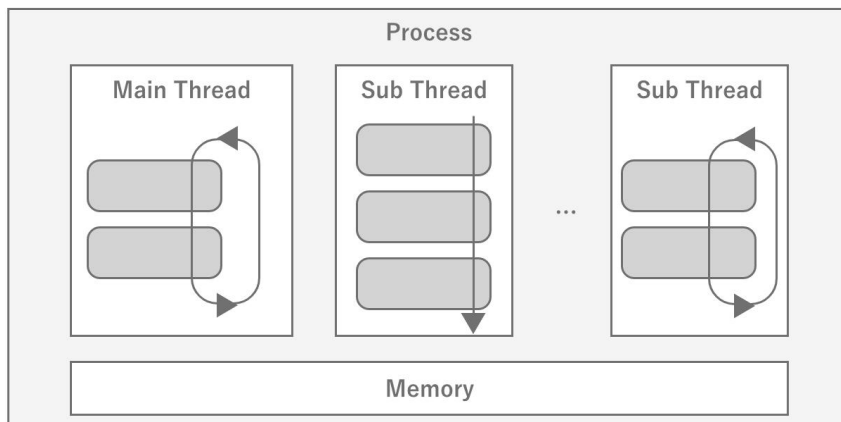
Also, since the asset entity is on the native side, the difficulty of investigating mul-

^{*3} <https://github.com/Unity-Technologies/UnityCsReference>

multiple asset loads and leaks increases. This is because developers mainly focus on profiling and debugging the C# side. It is difficult to understand the C# side execution state alone, and it is necessary to analyze it by comparing it with the engine side execution state. Profiling of the native area is dependent on the API provided by Unity, which limits the tools available. We will introduce methods for analysis using a variety of tools in this document, but it will be easier to understand if you are aware of the space between C# and native.

2.4.3 Threads

A thread is a unit of program execution, and processing generally proceeds by creating multiple threads within a single process. Since a single core of the CPU can only process one thread at a time, it executes the program while switching between threads at high speed to handle multiple threads. This is called **context switch** and is called a context switch. Context switches incur overhead, so if they occur frequently, processing efficiency is reduced.



▲ Figure 2.31 Schematic diagram of a thread

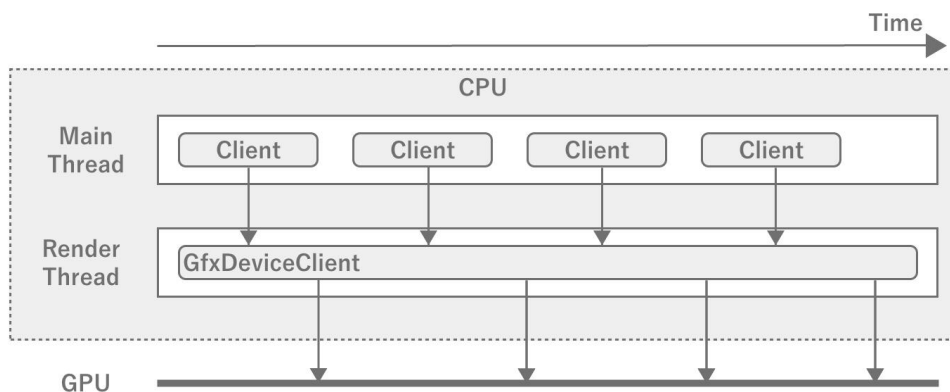
When a program is executed, the underlying **main thread** is created, from which the program creates and manages other threads as needed. Unity's game loop is designed to run on a single thread, so scripts written by users will basically run on the main thread. Conversely, attempting to call Unity APIs from a thread other than

the main thread will result in an error for most APIs.

If you create another thread from the main thread to execute a process, you do not know when that thread will be executed and when it will complete. Therefore, the means to synchronize processing between threads is to use the **signal** mechanism to synchronize processing between threads. When a thread is waiting for another thread to complete a process, it can be released by receiving a signal from that thread. This signal waiting is also used within Unity and can be observed during profiling, but it is important to note that it is just waiting for another process, as the name `WaitFor~` implies.

Threads inside Unity

However, if every process is running in the main thread, the entire program will take a long time to process. If there are multiple heavy processes and they are not interdependent, it is possible to shorten program execution if parallel processing can be done by synchronizing the processes to some extent. To achieve such speed, a number of parallel processes are used inside the game engine. One of them is **Render Thread** one of them is the render thread. As the name suggests, it is a thread dedicated to rendering and is responsible for sending frame drawing information calculated by the main thread to the GPU as graphics commands.



▲ Figure 2.32 Main Thread and Render Thread

The main thread and the render thread run like a pipeline, so the render thread starts computing the next frame while the render thread is processing it. However, if

the time to process a frame in the render thread is getting longer, it will not be able to start drawing the next frame even if the calculation for the next frame is finished, and the main thread will have to wait. In game development, be aware that the FPS will drop if either the main thread or the render thread becomes too heavy.

Parallelizable User Processing Threading

In addition, there are many calculation tasks that can be executed in parallel, such as physics engine and shaking, which are unique to games. In order to execute such calculations outside of the main thread, Unity uses the **Worker Thread** (Worker Thread) exists to execute such calculations outside of the main thread. Worker threads execute computation tasks generated through the JobSystem. If you can reduce the processing load on the main thread by using JobSystem, you should actively use it. Of course, you can also generate your own threads without using JobSystem.

While threads are useful for performance tuning, we recommend that you do not use them in the dark, since there is a risk that using too many of them may conversely degrade performance and increase the complexity of processing.

2.4.4 Game Loop

Common game engines, including Unity, use the **Game Loop** (Player Loop) which is a routine process of the engine. A simple way to describe the loop is roughly as follows

1. Processing input from controllers such as keyboard, mouse, touch display, etc.
2. Calculating the game state that should progress in one frame of time
3. Rendering the new game state
4. Waiting until the next frame depending on the target FPS

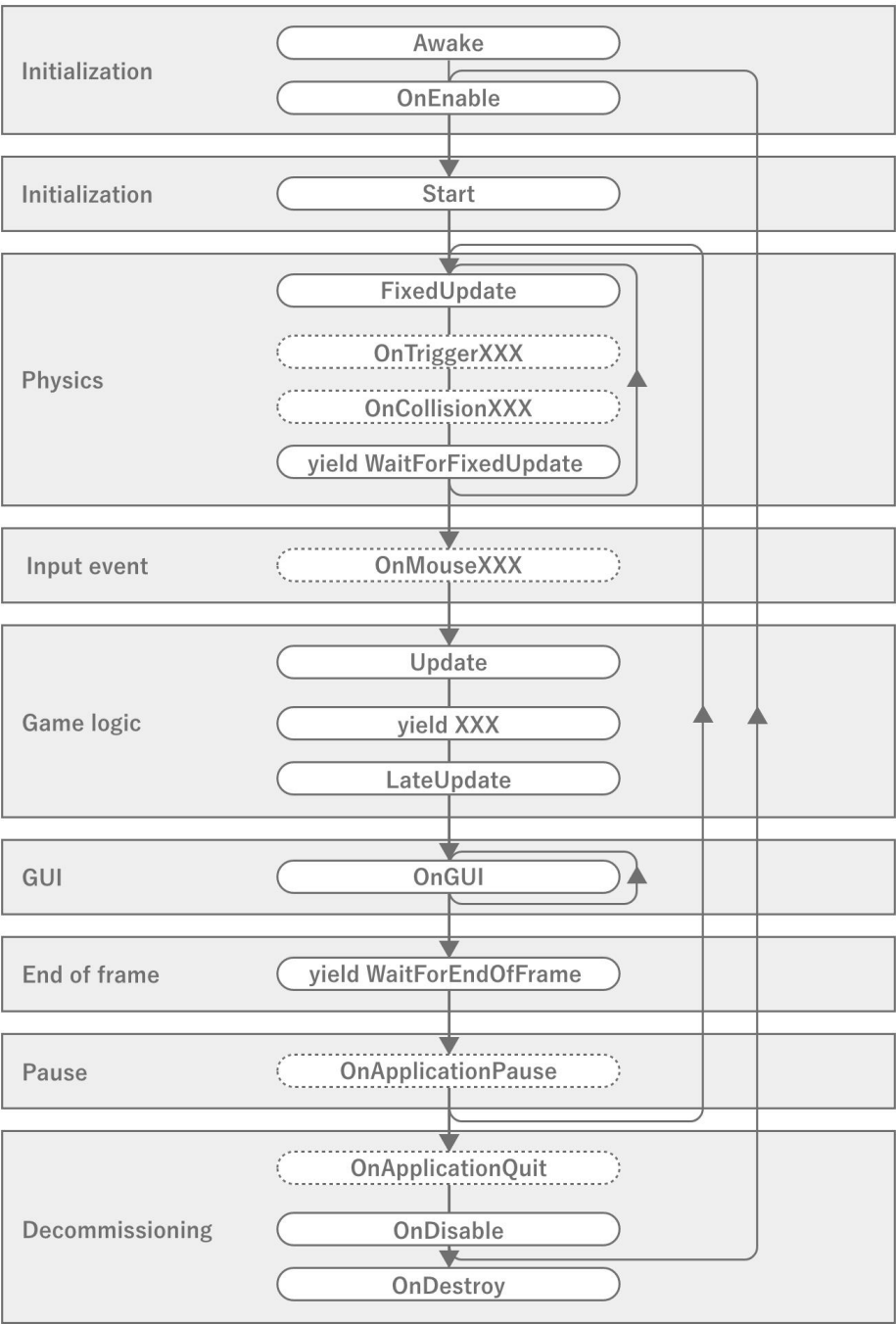
This loop is repeated to output the game as a video to the GPU. If processing within a single frame takes longer, then of course the FPS will drop.

Chapter 2 Fundamentals

Game Loop in Unity

The game loop in Unity is illustrated in the official Unity reference ^{*4}, which you may have seen at least once.

^{*4} <https://docs.unity3d.com/ja/current/Manual/ExecutionOrder.html>



▲ Figure 2.33 Event Execution Order in Unity

This diagram strictly shows the order of execution of events in `MonoBehaviour`, which is different from the game engine game loop^{*5}, but is sufficient for a game loop that developers should know. Especially important are the events `Awake`, `OnEnable`, `Start`, `FixedUpdate`, `Update`, `LateUpdate`, `OnDisable`, `OnDestroy` and the timing of the various coroutines. Mistaking the order of execution or timing of events can lead to unexpected memory leaks or extra calculations. Therefore, you should be aware of the nature of important event call timing and the order of execution within the same event.

There are some specific problems with physics calculations, such as objects slipping through without being detected as collisions if they are executed at the same intervals as in the normal game loop. For this reason, the physics routines are usually looped at different intervals from the game loop so that they are executed at a high frequency. However, if the loops are run at a very fast interval, there is a possibility that they will conflict with the update process of the main game loop, so it is necessary to synchronize the processes to a certain extent. Therefore, be aware of the possibility that the physics operations may affect the frame drawing process if the physics operations are heavier than necessary, or that the physics operations may be delayed and slip through if the frame drawing process is heavier.

2.4.5 GameObject

As mentioned above, since the Unity engine itself runs natively, the Unity API in C# is also, for the most part, an interface for calling the internal native API. The same is true for `GameObject` and `MonoBehaviour`, which defines components that attach to it, which will always have native references from the C# side. However, if the native side manages the data and also has references to them on the C# side, there is an inconvenience when it comes time to destroy them. This is because the references from C# cannot be deleted without permission when the data is destroyed on the native side.

In fact, List 2.1 checks if the destroyed `GameObject` is null, but `true` is output in the log. This is unnatural for standard C# behavior, since `_gameObject` is not assigned null, so there should still be a reference to an instance of type `GameObject`.

^{*5} <https://tsubakit1.hateblo.jp/entry/2018/04/17/233000>

▼ List 2.1 Post-destruction reference test

```
public class DestroyTest : UnityEngine.MonoBehaviour
{
    private UnityEngine.GameObject _gameObject;

    private void Start()
    {
        _gameObject = new UnityEngine.GameObject("test");
        StartCoroutine(DelayedDestroy());
    }

    System.Collections.IEnumerator DelayedDestroy()
    {
        // cache WaitForSeconds to reuse
        var waitOneSecond = new UnityEngine.WaitForSeconds(1f);
        yield return waitOneSecond;

        Destroy(_gameObject);
        yield return waitOneSecond;

        // _gameObject is not null, but result is true
        UnityEngine.Debug.Log(_gameObject == null);
    }
}
```

This is because Unity's C# side mechanism controls access to destroyed data. In fact, if you refer to the source code⁶ of `UnityEngine.Object` in Unity's C# implementation section, you will see the following

▼ List 2.2 `UnityEngine.Object`'s `==` operator implementation

```
// Excerpt.
public static bool operator==(Object x, Object y) {
    return CompareBaseObjects(x, y);
}

static bool CompareBaseObjects(UnityEngine.Object lhs,
    UnityEngine.Object rhs)
{
    bool lhsNull = ((object)lhs) == null;
    bool rhsNull = ((object)rhs) == null;

    if (rhsNull && lhsNull) return true;

    if (rhsNull) return !IsNativeObjectAlive(lhs);
    if (lhsNull) return !IsNativeObjectAlive(rhs);

    return lhs.m_InstanceID == rhs.m_InstanceID;
}
```

⁶ <https://github.com/Unity-Technologies/UnityCsReference/blob/c84064be69f20dcf21ebe4a7bbc176d48e2f289c/Runtime/Export/Scripting/UnityEngineObject.bindings.cs>

```
    }

    static bool IsNativeObjectAlive(UnityEngine.Object o)
    {
        if (o.GetCachedPtr() != IntPtr.Zero)
            return true;

        if (o is MonoBehaviour || o is ScriptableObject)
            return false;

        return DoesObjectWithInstanceIDExist(o.GetInstanceID());
    }
```

To summarize, a null comparison to a destroyed instance is `true` because when a null comparison is made, the native side is checked to see if the data exists. This causes instances of `GameObject` that are not null to behave as if they are partially null. While this characteristic is convenient at first glance, it also has a very troubling aspect. That is because `_gameObject` is not actually null, which causes a memory leak. A memory leak for a single `_gameObject` is obvious, but if you have a reference to a huge piece of data, for example a master, from within that component, it will lead to a huge memory leak because the reference remains as C# and is not subject to garbage collection. To avoid this, you need to take measures such as assigning null to `_gameObject`.

2.4.6 AssetBundle

Games for smartphones are limited by the size of the app, and not all assets can be included in the app. Therefore, in order to download assets as needed, Unity has a mechanism called `AssetBundle` that packs multiple assets and loads them dynamically. At first glance, this may seem easy to handle, but in a large project, it requires careful design and a good understanding of memory and `AssetBundle`, as memory can be wasted in unexpected places if not designed properly. Therefore, this section describes what you need to know about `AssetBundle` from a tuning perspective.

Compression settings for AssetBundle

`AssetBundle` is LZMA compressed by default at build time. This can be changed to uncompressed by changing `BuildAssetBundleOptions` to `UncompressedAssetBundle` and to LZ4 compression by changing to `ChunkBasedCompression`. The difference between these settings tends to look like the following Table 2.4

▼ Table 2.4 Differences between AssetBundle compression settings

Item	Uncompressed	LZMA	LZ4
File size	extra large	Extra Small	small
Load Time	fast	slow	Fairly fast

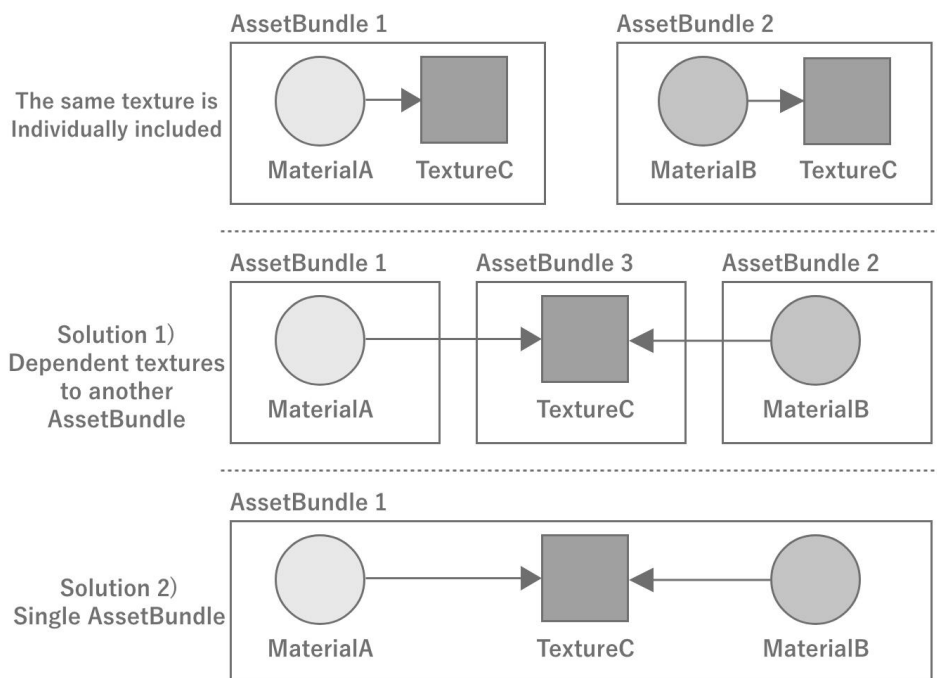
In other words, uncompressed is good for the fastest loading time, but its fatally large file size makes it basically unusable to avoid wasting storage space on smartphones. LZMA, on the other hand, has the smallest file size, but has the disadvantages of slow decompression and partial decompression due to algorithm problems. LZ4 is a compression setting that offers a good balance between speed and file size, and as the name `ChunkBasedCompression` suggests, partial decompression is possible, so partial loading is possible without having to decompress the entire file as with LZMA.

AssetBundle also has `Caching.compressionEnabled`, which changes the compression settings when cached in the terminal cache. In other words, by using LZMA for delivery and converting to LZ4 on the terminal, the download size can be minimized and the benefits of LZ4 can be enjoyed when actually used. However, recompression on the terminal side means that the CPU processing cost on the terminal is that much higher, and memory and storage space are temporarily wasted.

AssetBundle Dependencies and Duplication

If an asset is dependent on multiple assets, care must be taken when converting it to an AssetBundle. For example, if material A and material B depend on texture C, and you create an AssetBundle for material A and B without creating an AssetBundle for the texture, the two AssetBundles generated by will each contain texture C, which will result in duplication and waste. This would be wasteful in terms of space usage. Of course, this is wasteful in terms of space usage, but it also wastes memory because the textures are instantiated separately when the two materials are loaded into memory.

To avoid having the same asset in multiple AssetBundles, texture C should be a standalone AssetBundle that is dependent on the material's AssetBundle, or Material A, B and texture C in a single AssetBundle. Material A, B and texture C must be made into a single AssetBundle.

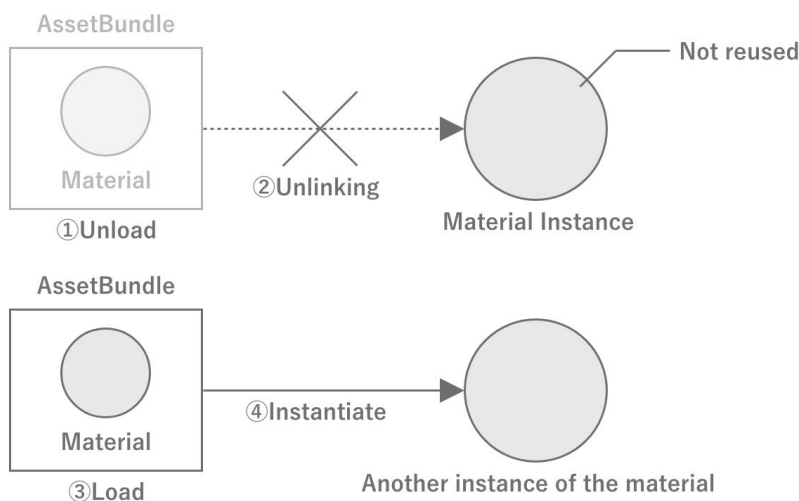


▲ Figure 2.34 Example with AssetBundle dependencies

Identity of assets loaded from AssetBundle

An important property of loading assets from an AssetBundle is that as long as the AssetBundle is loaded, the same instance of the same asset will be returned no matter how many times the asset is loaded. This indicates that Unity internally manages the loaded assets, and the AssetBundle and the assets are tied together within Unity. By using this property, it is possible to leave the caching of assets to Unity without creating a cache mechanism for them on the game side.

Note, however, that an asset unloaded at `AssetBundle.Unload(false)` will become a different instance even if the same asset is loaded again from the same AssetBundle as at Figure 2.35. This is because the AssetBundle is unlinked from the asset at the time of unloading, and the management of the asset is in a state of floating in the air.



▲ Figure 2.35 Example of memory leakage due to improper management of AssetBundle and assets

Destroying assets loaded from AssetBundle

When unloading AssetBundle using `AssetBundle.Unload(true)`, the loaded assets are completely discarded, so there is no memory problem. However, when using `AssetBundle.Unload(false)`, the assets are not discarded unless the asset unload instruction is called at the appropriate time. Therefore, when using the latter, it is necessary to call `Resources.UnloadUnusedAssets` appropriately so that assets are destroyed when switching scenes, etc. Also note that, as the name `Resources.UnloadUnusedAssets` implies, if a reference remains, it will not be released. Note that when `Addressable` is used, `AssetBundle.Unload(true)` is called internally.

2.5 C# Basics

This section describes the language specification and program execution behavior of C#, which is essential for performance tuning.

2.5.1 Stack and Heap

"Stack and Heap" introduced the existence of stack and heap as memory management methods during program execution. The stack is managed by the OS, while the heap is managed by the program. In other words, knowing how heap memory is managed allows for memory-aware implementation. Since the mechanism for managing heap memory depends largely on the language specification of the source code from which the program originates, we will explain heap memory management in C#.

Heap memory is allocated when necessary and must be released when it is finished being used. If memory is not released, a memory leak occurs and the memory area used by the application expands, eventually leading to a crash. C#, however, does not have an explicit memory release process. NET runtime environment in which C# programs are executed, heap memory is automatically managed by the runtime, and memory that has been used up is released at the appropriate time. For this reason, heap memory is referred to as **Managed Heap** is also referred to as managed heap memory.

The memory allocated on the stack matches the lifetime of the function, so it only needs to be released at the end of the function. heap memory allocated on the heap will most likely survive beyond the lifetime of the function, meaning that heap memory is used only when the function finishes using it. This means that heap memory is needed and used at different times, so a mechanism is needed to use heap memory automatically and efficiently. The details are presented in the next section. **Garbage Collection** Garbage Collection

In fact, Unity's **Alloc** is a proprietary term that refers to the memory allocated to the heap memory managed by garbage collection. Therefore, reducing GC.Alloc will reduce the amount of heap memory allocated dynamically.

2.5.2 Garbage Collection

In C# memory management, the search and release of unused memory is called garbage collection, or "GC" for short. The garbage collector is executed cyclically. However, the exact timing of execution depends on the algorithm. It performs a simultaneous search of all objects on the heap and deletes all objects that are already

dereferenced. In other words, dereferenced objects are deleted, freeing up memory space.

There are various algorithms for garbage collectors, but Unity uses the Boehm GC algorithm by default. The Boehm GC algorithm is characterized by being "non-generational" and "uncompressible. Non-generation-specific" means that the entire heap has to be scanned at once for each garbage collection run. This reduces performance because the search area expands as the heap expands. Uncompressed" means that objects are not moved in memory to close gaps between objects. This means that fragmentation, which creates small gaps in memory, tends to occur and the managed heap tends to expand.

Each is a computationally expensive and synchronous process that stops all other processing, leading to the so-called "Stop the World" process drop when running during a game.

Starting with Unity 2018.3, GCMode can be specified and can be temporarily disabled.

```
1: GarbageCollector.GCMode = GarbageCollector.Mode.Disabled;
```

But of course, if GC.Alloc is done during the period of disabling, the heap space will be extended and consumed, eventually leading to a crash of the app as it cannot be newly allocated. Since memory usage can easily increase, it is necessary to implement the function so that GC.Alloc is not performed at all during the period when it is disabled, and the implementation cost is also high, so the actual use is limited. (e.g., disabling only the shooting part of a shooting game)

In addition, Incremental GC can be selected starting with Unity 2019. With Incremental GC, garbage collection processing is now performed across frames, and large spikes can now be reduced. However, for games that must maximize power while reducing processing time per frame, it is necessary to implement an implementation that avoids the occurrence of GC.Alloc when it comes down to it. Specific examples are discussed at "10.1 GC.Alloc cases and how to deal with them".

When should we start working on this?

Because of the large amount of code in a game, if performance tuning is performed after the implementation of all functions is complete, you will often encounter designs/implementations that do not avoid GC.Alloc. If you are always aware of where it occurs from the initial design stage before coding, the cost of rework can be reduced, and total development efficiency tends to improve.

The ideal implementation flow is to first create a prototype with an emphasis on speed to verify the feel and the core of the game. Then, when moving on to the next phase of production, the design is reviewed and restructured once again. During this restructuring phase, it would be healthy to work on eliminating GC.Alloc. In some cases, it may be necessary to reduce the readability of the code in order to speed up the process, so if we start from the prototype, the development speed will also decrease.

2.5.3 Structure (struct)

In C#, there are two types of composite type definitions: classes and structs. The basic premise is that classes are reference types and structs are value types. Citing MSDN's "Choosing Between Class and Struct" ^{*7}, we will review the characteristics of each, the criteria by which they should be chosen, and notes on their usage.

Differences in Memory Allocation Locations

The first difference between reference and value types is that they allocate memory differently. Although somewhat imprecise, it is safe to recognize the following. Reference types are allocated in the heap area of memory and are subject to garbage collection. Value types are allocated in the stack area of memory and are not subject to garbage collection. Allocation and deallocation of value types is generally less expensive than for reference types.

However, value types and static variables declared in fields of reference types are allocated in the heap area. Note that variables defined as structures are not necessarily allocated to the stack area.

^{*7} <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/choosing-between-class-and-struct>

Handling Arrays

Arrays of value types are allocated inline, and the array elements are the entities (instances) of the value type. On the other hand, in an array of reference type, the array elements are arranged by reference (address) to the entity of the reference type. Therefore, allocation and deallocation of arrays of value types is much less expensive than for reference types. In addition, in most cases, arrays of value types have the advantage that the locality (spatial locality) of references is greatly improved, which makes CPU cache memory hit probability higher and facilitates faster processing.

Value Copying

In reference-type assignment (allocation), the reference (address) is copied. On the other hand, in a value type assignment (allocation), the entire value is copied. The size of the address is 4 bytes in a 32-bit environment and 8 bytes in a 64-bit environment. Therefore, a large reference type assignment is less expensive than a value type assignment that is larger than the address size.

Also, in terms of data exchange (arguments and return values) using methods, the reference type passes the reference (address) by value, whereas the value type passes the instance itself by value.

```
1: private void HogeMethod(MyStruct myStruct, MyClass myClass){...}
```

For example, in this method, the entire value of `MyStruct` is copied. This means that as the size of `MyStruct` increases, so does the copy cost. On the other hand, the `MyClass` method only copies the reference to `myClass` as a value, so even if the size of `MyClass` increases, the copy cost will remain constant because it is only for the address size. Since the increase in copy cost is directly related to the processing load, the appropriate choice must be made according to the size of the data to be handled.

Immutability

Changes made to an instance of a reference type will affect other locations that reference the same instance. On the other hand, a copy of an instance of a value

type is created when it is passed by value. If an instance of a value type is modified, naturally does not affect the copy of that instance. The copy is not created explicitly by the programmer, but implicitly when the argument is passed or the return value is returned. As a programmer, you have probably experienced the bug at least once where you thought you were changing a value, but in fact you were just setting the value against the copy, which is not what was intended to do. It is recommended that value types be immutable, as changeable value types can be confusing to many programmers.

Pass-by-Reference

A common misapplication is that "reference types are always passed by reference," but as mentioned earlier, reference (address) copying is fundamental, and reference passing is done when the `ref/in/out` parameter modifier is used.

```
1: private void HogeMethod(ref MyClass myClass){...}
```

Since the reference (address) was copied in reference type value passing, replacing an instance does not affect the original instance, but reference passing allows replacing the original instance.

```
1: private void HogeMethod(ref MyClass myClass)
2: {
3:     // The original instance passed by argument is rewritten.
4:     myClass = new MyClass();
5: }
```

Boxing

Boxing is the process of converting a value type to a `object` type or a value type to an interface type. A box is an object that is allocated on the heap and subject to garbage collection. Therefore, an excess of boxing and unboxing will result in

GC.Alloc. In contrast, when a reference type is cast, no such boxings take place.

▼ List 2.7 When a value type is cast to an object type, boxed

```
1: int num = 0;
2: object obj = num; // Boxed
3: num = (int) obj; // Unboxing
```

We would never use such obvious and meaningless boxings, but what about when they are used in the method?

▼ List 2.8 Example of boxed by implicit cast

```
1: private void HogeMethod(object data){ ... }
2:
3: // Abbreviation
4:
5: int num = 0;
6: HogeMethod(num); // Boxing with arguments
```

Cases like this exist where boxings are unintentionally boxed.

Compared to simple assignment, boxing and unboxing is a burdensome process. When boxed value types are boxed, new instances must be allocated and constructed. Also, although not as burdensome as boxing, the casting required for unboxing is also very burdensome.

Criteria for Selecting Classes and Structures

- Conditions under which structures should be considered :
 - When instances of the type are often small and have a short validity period
 - When the type is often embedded in other objects
- Conditions for avoiding structures: unless the type has all of the following characteristics
 - When it logically represents a single value, as with primitive types (int, double, etc.)
 - The size of the instance is less than 16 bytes
 - It is immutable.
 - Does not need to be boxed frequently

There are a number of types that do not meet the above selection criteria but are defined as structures. Types such as `Vector4` and `Quaternion`, which are frequently

used in Unity, are defined as structs, though not less than 16 bytes. Please check how to handle these efficiently, and if copying costs are increasing, choose a method that includes a workaround. In some cases, consider creating an optimized version with equivalent functionality on your own.

2.6 Algorithms and computational complexity

Game programming uses a variety of algorithms. Depending on how the algorithm is created, the calculation result may be the same, but the performance may vary greatly due to differences in the calculation process. For example, you will want a metric to evaluate how efficient the standard C# algorithm is and how efficient your implementation of the algorithm is, respectively. As a guide to measure these, a measure called computational complexity is used.

2.6.1 About computational complexity

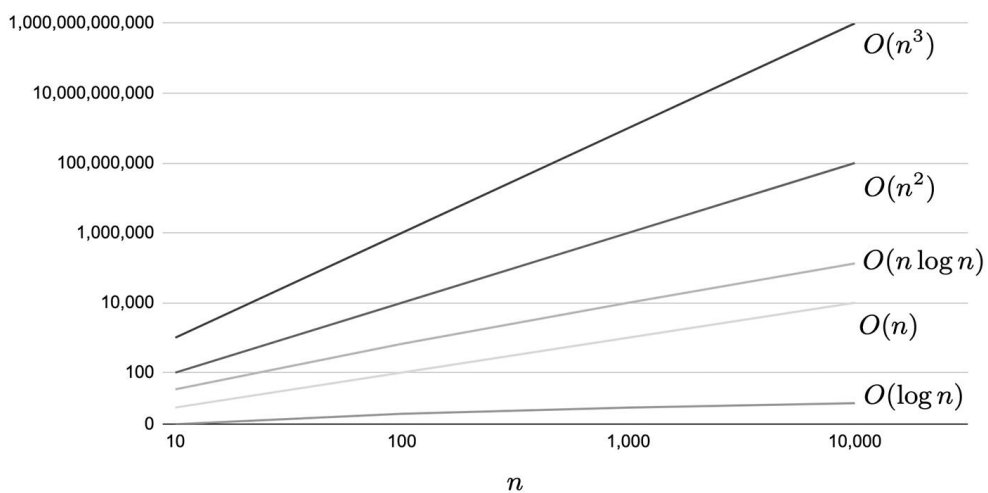
The computational complexity is a measure of an algorithm's computational efficiency, and can be subdivided into time complexity, which measures time efficiency, and area complexity, which measures memory efficiency. The order of computational complexity is O notation (Landau's symbol). Since computer science and mathematical definitions are not the essence here, please refer to other books if you are interested. In this paper, the quantity of calculations is treated as time-calculated quantities.

The main commonly used computational quantities are $O(1)$, $O(2)$, $O(3)$, $O(4)$, $O(n)$ and $O(n^2)$, $O(n \log n)$ are denoted as In parentheses n in parentheses indicates the number of data. It is easy to understand if you imagine how much the number of times a certain process is processed depends on the number of data. To compare performance in terms of computational complexity, see $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3)$ The result is as follows. Table 2.5 The comparison of the number of data and the number of calculation steps and the comparison graph displayed logarithmically at Figure 2.36 are shown in the following table. $O(1)$ is excluded because it does not depend on the number of data and is obviously more efficient than $O(1)$. For example, for $O(\log n)$ has 13 computation steps even if there are 10,000 samples, and has 23 computation steps even if there are 10 million samples, which shows that it is extremely superior.

2.6 Algorithms and computational complexity

▼ Table 2.5 Number of data and number of computation steps for major quantities

n	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$
10	3	10	33	100	1,000
100	7	100	664	10,000	1,000,000
1,000,000	10	1,000	9,966	1,000,000	1,000,000,000
10,000	13	10,000	132,877	100,000,000	1,000,000,000 1,000,000,000



▲ Figure 2.36 Comparison of performance differences in logarithmic representation of each calculation amount

To illustrate each of the computational quantities, we will list a few code samples. First, let's look at the following code samples. $O(1)$ indicates a constant amount of computation independent of the number of data.

▼ List 2.9 *Code example of $O(1)$* Code example of $O(1)$

```

1: private int GetValue(int[] array)
2: {
3:     // Assume that array is an array containing some integer value.
4:     var value = array[0];
5:     return value;
6: }

```

Aside from the *raison d'être* of this method, obviously the process is independent

Chapter 2 Fundamentals

of the number of data in the array and takes a constant number of calculations (in this case, one).

Next, we call $O(n)$ code example.

▼ List 2.10 $O(n)$ Code Example of $O(n)$

```
1: private bool HasOne(int[] array, int n)
2: {
3:     // Assume that array has length=n and contains some integer value
4:     for (var i = 0; i < n; ++i)
5:     {
6:         var value = array[i];
7:         if (value == 1)
8:         {
9:             return true;
10:        }
11:    }
12: }
```

Here is an array containing integer values with 1 is present, the process just returns true. If by chance the first of array1 is found at the beginning of , the process may be completed in the fastest possible time, but if there is no 1 in array , the process will return 1 or at the end of array for the first time, the process will return 1 is found for the first time at the end of , the loop will go all the way to the end, so n times because the loop goes all the way to the end. This worst-case scenario is called $O(n)$ and you can imagine that the amount of computation increases with the number of data.

Next, let us denote the worst-case scenario as $O(n^2)$ Let's look at an example for the case of $O(n^2)$.

▼ List 2.11 $O(n^2)$ Example code for

```
1: private bool HasSameValue(int[] array1, int[] array2, int n)
2: {
3:     // Assume array1 and array2 have length=n and contain some integer value.
4:     for (var i = 0; i < n; ++i)
5:     {
6:         var value1 = array1[i];
7:         for (var j = 0; j < n; ++j)
8:         {
9:             var value2 = array2[j];
10:            if (value1 == value2)
11:            {
12:                return true;
13:            }
14:        }
15:    }
16:
17:    return false;
```



```
18: }
```

This one is just a method that returns `true` if any of the two arrays contain the same value in a double loop. The worst-case scenario is that they are all mismatched cases, so n^2 times.

As a side note, in the concept of computational complexity, only the term with the largest order is used. If we create a method that executes each of the three methods in the above example once, we get the maximum order $O(n^2)$ of the maximum order. (The $O(n^2 + n + 1)$)

It should also be noted that the calculation volume is only a guideline when the number of data is sufficiently large, and is not necessarily linked to the actual measurement time. $O(n^5)$ may not be a problem when the number of data is small, even if it looks like a huge calculation volume such as . Therefore, it is recommended to use the calculation volume as a reference and measure the processing time to see if it fits within a reasonable range, taking the number of data into consideration each time.

2.6.2 Basic Collections and Data Structures

C# provides collection classes with various data structures. This section introduces the most frequently used ones as examples, and shows in what situations you should employ each of them, based on the computation time of the main methods.

The method complexity of the collection classes described here can be found on MSDN at . It is safer to check the MSDN when selecting the most appropriate collection class.

List<T>

This is the most commonly used `List<T>`. The data structure is an array. It is effective when the order of data is important, or when data is often retrieved or updated by index. On the other hand, if there are many insertions and deletions of elements, it is best to avoid using `List<T>` because it requires a large amount of computation

Chapter 2 Fundamentals

due to the need to copy after the indexes that have been manipulated.

In addition, when the capacity is exceeded by `Add`, the memory allocated for the array is extended. When memory is extended, twice the current Capacity is allocated, so it is recommended that `Add` be used with $O(1)$ to use it with appropriate initial values so that it can be used without causing expansion.

▼ Table 2.6 `List<T>`

Method	Calculation
<code>Add</code>	$O(1)$ However, if capacity is exceeded $O(n)$
<code>Insert</code>	$O(n)$
<code>IndexOf/Contains</code>	$O(n)$
<code>RemoveAt</code>	$O(n)$
<code>Sort</code>	$O(n \log n)$

LinkedList<T>

The data structure of `LinkedList<T>` is a linked list. A linked list is a basic data structure, in which each node has a reference to the next node. C#'s `LinkedList<T>` is a two-way linked list, so each has a reference to the node before and after it. `LinkedList<T>` has strong features for adding and deleting elements, but is not good at accessing specific elements in the array. It is suitable when you want to create a process that temporarily holds data that needs to be added or deleted frequently.

▼ Table 2.7 `LinkedList<T>`

Method	Computation
<code>AddFirst/AddLast</code>	$O(1)$
<code>AddAfter/AddBefore</code>	$O(1)$
<code>Remove/RemoveFirst/RemoveLast</code>	$O(1)$
<code>Contains</code>	$O(n)$

Queue<T>

`Queue<T>` is a collection class that implements the FIFO (first in first out) method. It is used to implement so-called queues, for example, to manage input operations. In `Queue<T>`, a circular array is used. `Enqueue` The first element is added at the end with `Dequeue` and the first element is removed while the second element is removed with `Dequeue`. When adding beyond capacity, expansion is performed. `Peek` is an operation to take out the top element without deleting it. As you can see from the computational

2.6 Algorithms and computational complexity

complexity, `Enqueue` and `Dequeue` can be used to keep high performance, but they are not suitable for operations such as traversal. `TrimExcess` is a method to reduce capacity, but from a performance tuning perspective, it can be used so that capacity is not increased or decreased in the first place, further exploiting `Queue<T>` its strengths.

▼ Table 2.8 `Queue<T>`

Method	Compute capacity
<code>Enqueue</code>	$O(1)$ However, if capacity is exceeded $O(n)$
<code>Dequeue</code>	$O(1)$
<code>Peek</code>	$O(1)$
<code>Contains</code>	$O(n)$
<code>TrimExcess</code>	$O(n)$

`Stack<T>`

`Stack<T>` is a collection class that implements the last in first out (LIFO) method: last in first out. `Stack<T>` is implemented as an array. `Push` The first element is added with `Pop`, and the first element is removed with `Pop`. `Peek` is an operation to take out the first element without deleting it. A common use of is when implementing screen transitions, where the scene information for the destination of the transition is stored in `Push`, and when the back button is pressed, retrieving the scene information by `Pop`. As with `Queue`, high performance can be obtained by using only `Push` and `Pop` for `Stack`. Be careful not to search for elements, and be careful to increase or decrease capacity.

▼ Table 2.9 `Stack<T>`

Method	Compute capacity
<code>Push</code>	$O(1)$ However, if capacity is exceeded $O(n)$
<code>Pop</code>	$O(1)$
<code>Peek</code>	$O(1)$
<code>Contains</code>	$O(n)$
<code>TrimExcess</code>	$O(n)$

`Dictionary<TKey, TValue>`

While the collections introduced so far have been semantic in order, `Dictionary<TKey, TValue>` is a collection class that specializes in indexability. The data structure is implemented as a hash table (a kind of associative array). The structure is like a

dictionary where keys have corresponding values (in the case of a dictionary, words are keys and descriptions are values). `Dictionary<TKey, TValue>` has the disadvantage of consuming more memory, but the speed of the lookup is $O(1)$ and faster. It is very useful for cases that do not require enumeration or traversal, and where the emphasis is on referencing values. Also, be sure to pre-set the capacity.

▼ Table 2.10 `Dictionary<TKey, TValue>`

Method	Compute Capacity
Add	$O(1)$ However, if capacity is exceeded $O(n)$
TryGetValue	$O(1)$
TryGetValue	$O(1)$
ContainsKey	$O(1)$
ContainsValue	$O(n)$

2.6.3 Devices to Lower the Calculation Volume

In addition to the collections introduced so far, various others are available. Of course, it is possible to implement the same process using only `List<T>` (array), but by selecting a collection class more suitable for , it is possible to optimize the amount of computation. By simply implementing methods with an awareness of the amount of computation, heavy processing can be avoided. As a way to optimize your code, you may want to check the computational complexity of your methods and see if you can reduce it to less than .

Means of devising: memoization

Suppose you have a method (`ComplexMethod`) with a very high computational complexity that requires complex calculations. However, there are times when it is not possible to reduce the amount of calculation. In such cases, a technique called memoization can be used.

Let us assume that `ComplexMethod` uniquely returns the corresponding result when given an argument. First, the first time the argument is passed, a complex process is passed through. After the calculation, the arguments and the result are put into `Dictionary<TKey, TValue>` and cached. The second

2.6 Algorithms and computational complexity

and subsequent times, we first check to see if they are cached, and if they are, we return only the result and exit. In this way, no matter how high the computation volume may be the first time, the second and subsequent times the computation volume is reduced to $O(1)$ the second time. If the number of arguments that can be passed is known in advance, it is possible to complete the calculation before the game and cache it, so that effectively returns $O(1)$ and cache them before the game.



PERFORMANCE TUNING BIBLE

CHAPTER

03

第3章

Profiling Tools

CyberAgent Smartphone Games & Entertainment

Chapter 3

Profiling Tools

Profiling tools are used to collect and analyze data, identify bottlenecks, and determine performance metrics. There are several of these tools provided by the Unity engine alone. Other tools include native-compliant tools such as Xcode and Android Studio, and GPU-specific tools such as RenderDoc. Therefore, it is important to understand the features of each tool and choose appropriately. In this chapter, we will introduce each tool and discuss profiling methods, aiming to help you use each tool appropriately.

3.0.1 Points to keep in mind when measuring

Since Unity can run applications on the editor, measurements can be taken both on the actual device and in the editor. It is necessary to keep in mind the characteristics of each environment when performing measurements.

The greatest advantage of using the editor is that it allows for quick trial and error. However, since the processing load of the editor itself and the memory area used by the editor are also measured, there will be a lot of noise in the measurement results. Also, since the specifications are completely different from those of the actual equipment, it is difficult to identify bottlenecks and the results may differ.

For this reason, we recommend that profiling be done on **the actual device**. However, it is efficient to complete the work only with the editor, which is less expensive, only when "it occurs in both environments. Most of the time, the problem is reproduced in both environments, but in rare cases, it may only be reproduced in one of the environments. Therefore, first confirm the phenomenon on the actual device. Next, it is recommended to confirm that the problem is reproduced in the editor as well, and then correct it in the editor. Of course, be sure to check the correction on the actual device at the end.

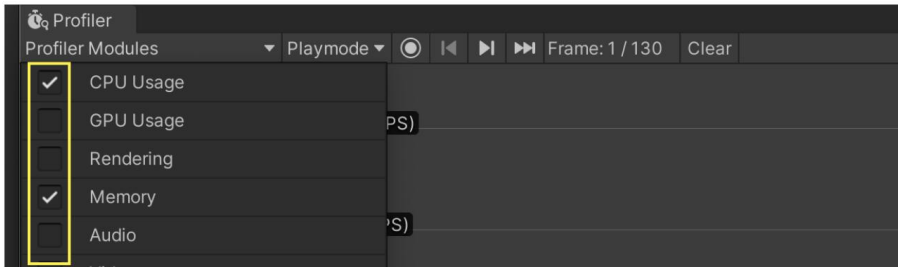
3.1 Unity Profiler

The Unity Profiler is a profiling tool built into the Unity Editor. This tool can collect information on a frame-by-frame basis. There is a wide range of items that can be measured, each called a profiler module, and in the Unity 2020 version there are 14 of them. This module is still being updated, and in Unity 2021.2, a new module on Asset and a new module on File I/O have been added. The Unity Profiler is a great tool for getting a rough look at performance because of the variety of modules available. The list of modules can be found at Figure 3.1.

Module Name	Description
CPU Usage	Breakdown of time spent by the CPU, including scripts and animations
GPU Usage	Breakdown of time spent by the GPU, including object rendering
Rendering	Information related to drawing, such as SetPass and Batching
Memory	Information on application-wide memory allocation
Audio	Memory allocation, CPU utilization, and other information about Audio
Video	Memory allocation about Video
Physics	Objects related to the physics engine
Physics2D	Object information about the 2D physics engine (RigidBody2D)
Network Messages (Deprecated)	Information on sending and receiving information about the (deprecated) Multiplayer High-Level API
Network Operations (Deprecated)	Information on sending and receiving information about the multiplayer low-level API (deprecated)
UI	UI-related processing time information
UI Details	Information on the number of batches and vertices in the UI display
Global Illumination	Information on time spent on GI lighting, including light probes
Virtual Texturing	Information about the Streaming Virtual Texturing feature
Asset Loading (2021.2 or later)	Texture, Mesh, and other information such as load timing, size, etc.
File Access (2021.2 or later)	Information related to file access, such as time spent on I/O

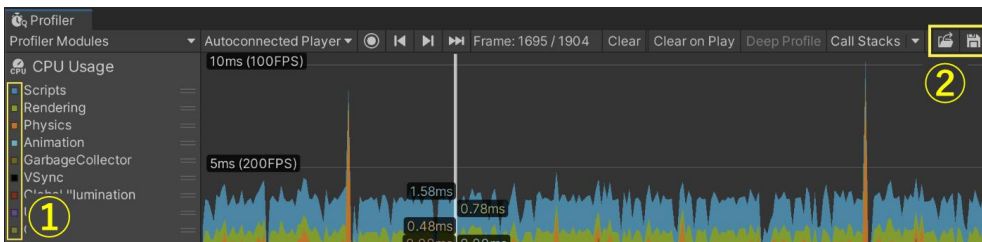
▲ Figure 3.1 List of Profiler Modules

These modules can be configured to be displayed or not on the profiler. However, modules that are not displayed are not measured. Conversely, if all of them are displayed, the editor will be overloaded.



▲ Figure 3.2 Show/Hide Function of Profiler Modules

The following are also useful functions common to the entire Profiler tool.



▲ Figure 3.3 Explanation of Profiler Functions

Figure 3.3 In the "Profiler Modules" section, "①" lists the items that each module is measuring. By clicking on this item, you can switch between display and non-display on the timeline on the right. Displaying only the necessary items will make the view easier to read. You can also reorder the items by dragging them, and the graph on the right side will be displayed in that order. The second item (2) is a function for saving and loading the measured data. It is recommended to save the measurement results if necessary. Only the data displayed on the profiler can be saved.

This book explains CPU Usage and Memory module, which are frequently used in Figure 3.1.

3.1.1 Measurement Methods

This section covers measurement methods using the Unity Profiler on an actual device. We will explain the measurement method in two parts, one before building and the other after launching the application. The measurement method in the editor is

Chapter 3 Profiling Tools

simply to click the measurement button during execution, so the details are omitted.

Work to be done before building

The work to be done before building is **Development Build** is to enable the "Development Build" setting. Once this is activated, a connection to the profiler can be established.

Also, we will need to enable the **Deep Profile** option for more detailed measurement. When this option is enabled, the processing time of all function calls is recorded, making it easier to identify bottleneck functions. The disadvantage is that the measurement itself requires a very large overhead, making it slow and memory intensive. Note that the process may appear to take a very long time, but not so much in the normal profile. Basically, it is used only when the normal profile does not provide enough information.

If Deep Profile uses a lot of memory, such as in a large project, it may not be possible to make measurements due to insufficient memory. In that case, you have no choice but to add your own measurement process by referring to "Supplement: About Sampler" in the "3.1.2 CPU Usage" section.

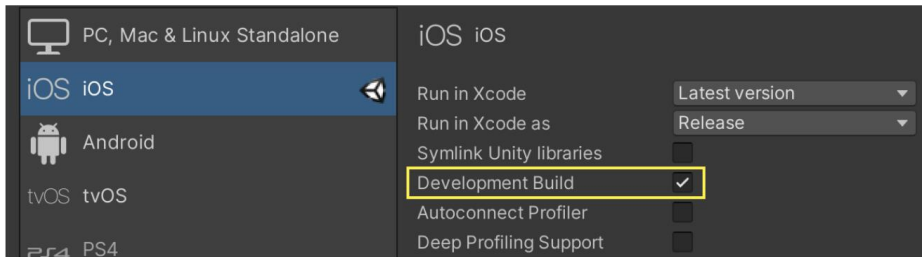
There are two ways to configure these settings: by explicitly specifying them in a script or by using the GUI. First, we will introduce the method of setting from a script.

▼ List 3.1 How to set up Development Build from a script

```
1: BuildPlayerOptions buildPlayerOptions = new BuildPlayerOptions();
2: /* Scene and build target settings are omitted. */
3:
4: buildPlayerOptions.options |= BuildOptions.Development;
5: // Add only if you want to enable Deep Profile mode
6: buildPlayerOptions.options |= BuildOptions.EnableDeepProfilingSupport;
7:
8: BuildReport report = BuildPipeline.BuildPlayer(buildPlayerOptions);
```

List 3.1 The important point in `BuildOptions.Development` is to specify .

Next, if you want to set up from GUI, go to Build Settings and check Development Build like Figure 3.4 and build.



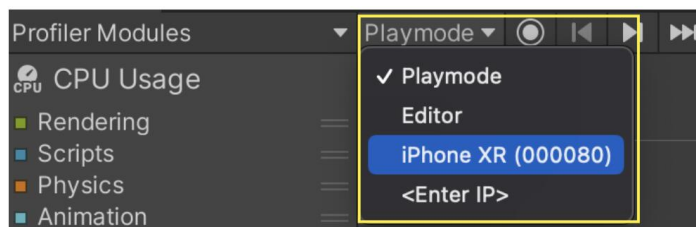
▲ Figure 3.4 Build Settings

Work to be done after application startup

There are two ways to connect with Unity Profiler after application startup: "Remote Connection" and "Wired (USB) Connection". The remote connection has more environmental restrictions than the wired connection, and the profile may not work as expected. For example, connection to the same Wifi network may be required, mobile communication may need to be disabled for Android only, and other ports may need to be freed. For this reason, this section will focus on wired connections, which are simpler and more reliable to profile. If you want to make a remote connection, please refer to the official documentation to give it a try.

First of all, for iOS, the procedure to connect to the profiler is as follows.

1. Change Target Platform to iOS from Build Settings
2. Connect the device to your PC and launch the Development Build application
3. Select the device to connect to from Unity Profiler (Figure 3.5)
4. Start Record



▲ Figure 3.5 Select the device to connect to

Chapter 3 Profiling Tools

The Unity Editor for measurement does not have to be the project you built. It is recommended to create a new project for the measurement, as it is lightweight.

Next, for Android, there are a few more steps than for iOS.

1. Change Target Platform to Android from Build Settings
2. Connect the device to the PC and launch the Development Build application
3. `adb forward` Enter the command. (Details of the command are described below.)
4. Select the device to connect to from Unity Profiler
5. Start Record

`adb forward` The command requires the Package Name of the application. For example, if the Package Name is "jp.co.sample.app", enter the following.

▼ List 3.2 `adb forward` command

```
1: adb forward tcp:34999 localabstract:Unity-jp.co.sample.app
```

If `adb` is not recognized, please set the `adb` path. There are many web-based instructions on how to set up `adb`, so we will skip this section.

For simple troubleshooting, if you cannot connect, check the following

- Common to both devices
 - Is there a "Development Build" sign in the lower right corner of the executed application?
- In the case of Android
 - Is USB debugging enabled on the device?
 - `adb forward` Is the package name entered in the command correct?
 - `adb devices` The device is properly recognized when the command is entered.

As an additional note, if you run the application directly in Build And Run, the `adb forward` command described above will be performed internally. Therefore, no command input is required for measurement.

Autoconnect Profiler

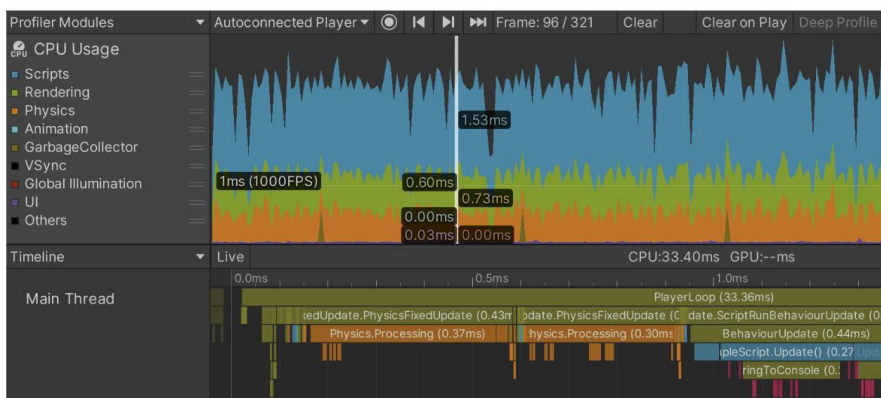
There is an Autoconnect Profiler option in the build configuration. This option is a function to automatically connect to the editor's profiler when the application is launched. Therefore, it is not a required setting for profiling. The same applies to remote profiling. Only WebGL cannot be profiled without this option, but it is not a very useful option for mobile.

To take this a bit further, if this option is enabled, the IP address of the editor will be written to the binary at build time, and an attempt will be made to connect to that address at startup. If you are building on a dedicated build machine, this is not necessary unless you are profiling on that machine. Rather, you will just have to wait longer (about 8 seconds) for the automatic connection to time out when the application starts.

Note that from the script, the option name is `BuildOptions.ConnectWithProfiler`, which can easily be mistaken for mandatory.

3.1.2 CPU Usage

CPU Usage is displayed as Figure 3.6.



▲ Figure 3.6 CPU Usage Module (Timeline Display)

Chapter 3 Profiling Tools

There are two main ways to check this module

- Hierarchy (Raw Hierarchy)
- Timeline

First, the Hierarchy view is explained in terms of what it shows and how to use it.

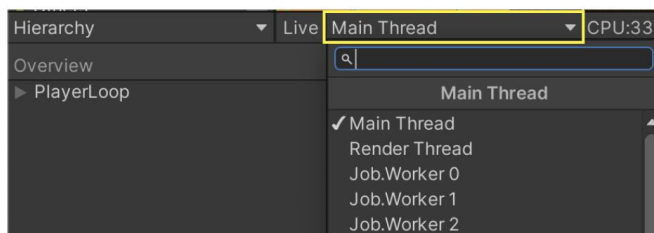
1. Hierarchy View

The Hierarchy view looks like Figure 3.7.

Hierarchy	Live	Main Thread	CPU:33.46ms	GPU:--ms	🔍	
Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ PlayerLoop	99.8%	0.1%	1	2.4 KB	33.42	0.06
▶ WaitForTargetFPS	88.5%	88.5%	1	0 B	29.63	29.62
▶ PostLateUpdate.FinishFrameRendering	7.7%	0.2%	1	0 B	2.59	0.07
▼ Update.ScriptRunBehaviourUpdate	1.5%	0.0%	1	2.4 KB	0.51	0.00
▼ BehaviourUpdate	1.5%	0.0%	1	2.4 KB	0.51	0.01
▶ SampleScript.Update()	1.2%	0.2%	2	2.4 KB	0.41	0.08
EventSystem.Update()	0.1%	0.1%	1	0 B	0.04	0.04
DebugUpdater.Update()	0.0%	0.0%	1	0 B	0.02	0.02

▲ Figure 3.7 Hierarchy View

This view is characterized by the fact that the measurement results are arranged in a list format and can be sorted by the items in the header. When conducting an investigation, bottlenecks can be identified by opening items of interest in the list. However, the information displayed is an indication of the time spent in the selected thread. For example, if you are using Job System or multi-threaded rendering, the processing time in another thread is not included. If you want to check, you can do so by selecting a thread like Figure 3.8.



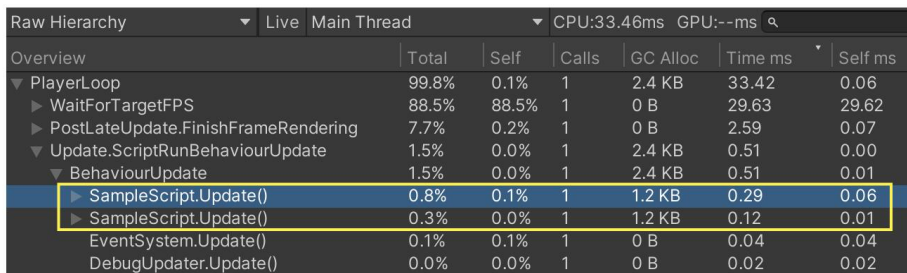
▲ Figure 3.8 Thread Selection

Next, the header items are explained.

▼ Table 3.1 Hierarchy Header Information

Header Name	Description
Overview	Sample name.
Total	Total time spent processing this function. (displayed as a percentage)
Self	Processing time of this function itself. Subfunction time is not included. (displayed in %) Self
Calls	Number of calls made in one frame.
GC Alloc	Heap memory allocated by this function.
Time ms	Total in ms.
Self ms	Self in ms.

Calls is easier to see as a view because it combines multiple function calls into a single item. However, it is not clear whether all of them have equal processing time or only one of them has a long processing time. In such cases, the **Raw Hierarchy View** is used in this case. The Raw Hierarchy view differs from the Hierarchy view in that Calls is always fixed at 1. Figure 3.9 In the following example, multiple calls to the same function are shown in the Raw Hierarchy view.



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ PlayerLoop	99.8%	0.1%	1	2.4 KB	33.42	0.06
▶ WaitForTargetFPS	88.5%	88.5%	1	0 B	29.63	29.62
▶ PostLateUpdate.FinishFrameRendering	7.7%	0.2%	1	0 B	2.59	0.07
▼ Update.ScriptRunBehaviourUpdate	1.5%	0.0%	1	2.4 KB	0.51	0.00
▼ BehaviourUpdate	1.5%	0.0%	1	2.4 KB	0.51	0.01
▶ SampleScript.Update()	0.8%	0.1%	1	1.2 KB	0.29	0.06
▶ SampleScript.Update()	0.3%	0.0%	1	1.2 KB	0.12	0.01
EventSystem.Update()	0.1%	0.1%	1	0 B	0.04	0.04
DebugUpdater.Update()	0.0%	0.0%	1	0 B	0.02	0.02

▲ Figure 3.9 Raw Hierarchy View

To summarize what has been said so far, the Hierarchy view is used for the following purposes

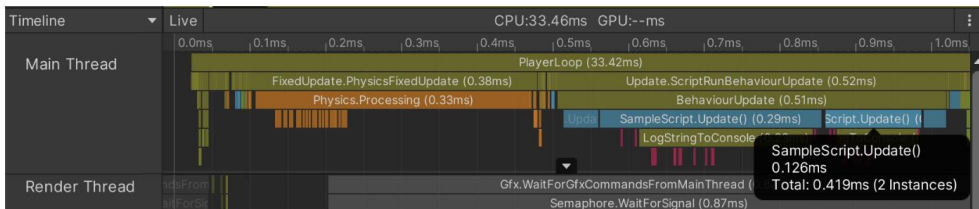
- Identify and optimize bottlenecks (Time ms, Self ms)
- Identify and optimize GC allocations (GC Allocation)

When performing these tasks, it is recommended to sort each desired item in descending order before checking it.

When opening an item, it is often the case that there is a deep hierarchy. In this case, you can open all levels of the hierarchy by holding down the Option key on a Mac (Alt key on Windows). Conversely, closing an item while holding down the key will close everything below that hierarchy.

2. timeline view

Another way to check the timeline view is as follows.



▲ Figure 3.10 Timeline View

In the timeline view, items in the hierarchy view are visualized as boxes, so you can intuitively see where the load is at a glance when viewing the entire view. And because it is mouse-accessible, even deep hierarchies can be grasped simply by dragging. In addition, with timelines, there is no need to switch threads; all threads are displayed. This makes it easy to see when and what kind of processing is taking place in each thread. Because of these features, timelines are mainly used for the following purposes

- To get a bird's eye view of the overall processing load
- To understand and tune the processing load of each thread

Timeline is not suited for sorting operations to determine the order of heavy processing, or for checking the total amount of allocations. Therefore, the Hierarchy View is better suited for tuning allocations.

Supplement: About Sampler

There are two ways to measure processing time per function. One is the Deep Profile mode described above. The other is to embed it directly in the script.

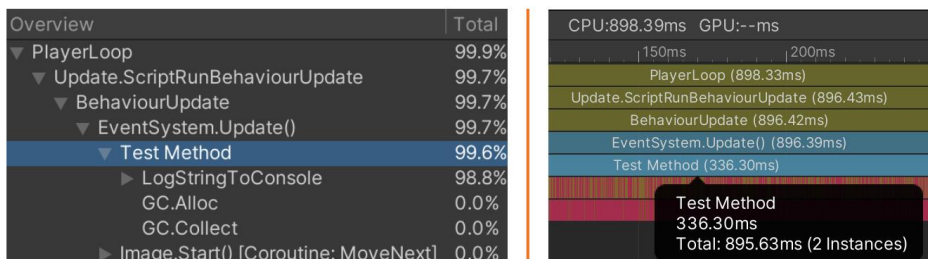
In the case of embedding directly in the script, use the following statement.

▼ List 3.3 Method using Begin/EndSample

```

1: using UnityEngine.Profiling;
2: /* ... Omitted...*/
3: private void TestMethod()
4: {
5:     for (int i = 0; i < 10000; i++)
6:     {
7:         Debug.Log("Test");
8:     }
9: }
10:
11: private void OnClickedButton()
12: {
13:     Profiler.BeginSample("Test Method")
14:     TestMethod();
15:     Profiler.EndSample()
16: }
```

The embedded sample will be displayed in both the Hierarchy and Timeline views.



▲ Figure 3.11 Sampler Display

There is one more feature worth mentioning. If the profiling code is not a Development Build, the caller is disabled, so there is zero overhead. It may be a good idea to put this in place in advance in areas where the processing load is likely to increase in the future.

The BeginSample method is a static function, so it can be used easily, but there is also a CustomSampler that has similar functionality. This method was added af-

Chapter 3 Profiling Tools

ter Unity 2017 and has less measurement overhead than `BeginSample`, so it can measure more accurate times.

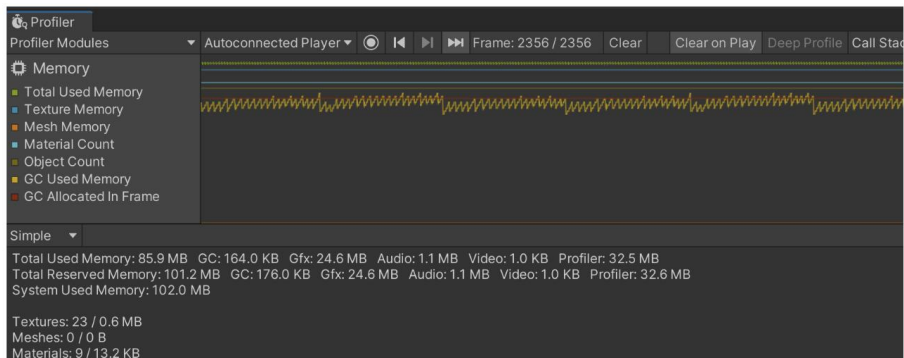
▼ List 3.4 How to use `CustomSampler`

```
1: using UnityEngine.Profiling;
2: /* ... Omitted...*/
3: private CustomSampler _samplerTest = CustomSampler.Create("Test");
4:
5: private void TestMethod()
6: {
7:     for (int i = 0; i < 10000; i++)
8:     {
9:         Debug.Log("Test");
10:    }
11: }
12:
13: private void OnClickedButton()
14: {
15:     _samplerTest.Begin();
16:     TestMethod();
17:     _samplerTest.End();
18: }
```

The difference is that an instance must be created in advance. Another feature of `CustomSampler` is that the measurement time can be obtained in the script after the measurement. If you need more accuracy or want to issue warnings based on processing time, `CustomSampler` is a good choice.

3.1.3 Memory

Memory modules are displayed as Figure 3.12.



▲ Figure 3.12 Memory Module

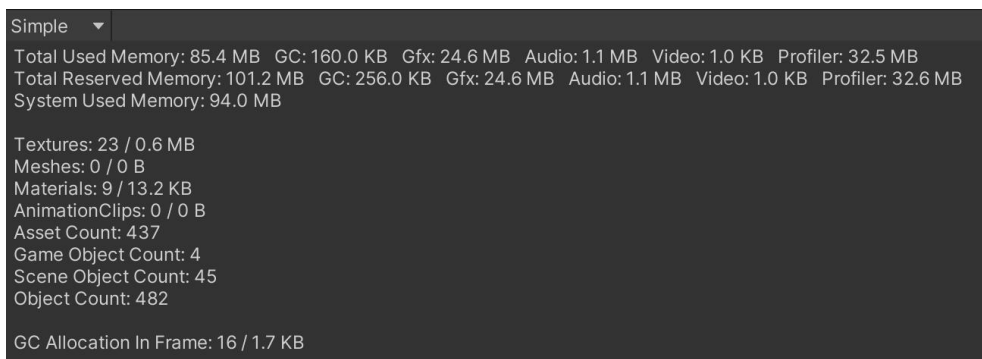
There are two ways to check this module

- Simple view
- Detailed view

First, we will explain the contents and usage of the Simple view.

1. simple view

The Simple view looks like Figure 3.13.



▲ Figure 3.13 Simple View

This section describes the items listed in the view.

Total Used Memory

Chapter 3 Profiling Tools

Total amount of memory allocated (in use) by Unity.

Total Reserved Memory

Total amount of memory currently reserved by Unity. A certain amount of contiguous memory space is reserved in advance by the OS as a pool, and is allocated when it is needed. When the pool area becomes insufficient, it is requested again from the OS side for expansion.

System Used Memory

Total amount of memory used by the application. This item also measures items (plug-ins, etc.) that are not measured in Total Reserved. However, it still does not track all memory allocations. To get an accurate picture, you will need to use a native-compliant profiling tool such as Xcode.

Figure 3.13 The meaning of the items listed to the right of Total Used Memory in

▼ Table 3.2 Simple View glossary

Term Name	Explanation
GC	Amount of memory used in the heap area. GC Alloc and other factors increase this amount.
Gfx	Amount of memory allocated for Texture, Shader, Mesh, etc.
Audio	Amount of memory used for audio playback.
Video	Amount of memory used for video playback.
Profiler	Amount of memory used for profiling.

As an additional note regarding the terminology names, starting with Unity 2019.2, "Mono" has been changed to "GC" and "FMOD" has been changed to "Audio".

Figure 3.13 The number of assets used and the amount of memory allocated for the following are also listed in the following table.

- Texture
- Mesh
- Material
- Animation Clip
- Audio Clip

The following information on the number of objects and GC Allocation is also available.

Asset Count

Total number of assets loaded.

Game Object Count

Number of game objects in the scene.

Scene Object Count

Total number of components and game objects in the scene.

Object Count

Total number of all objects generated and loaded by the application. If this value is increasing, it is likely that some objects are leaking.

GC Allocation in Frame

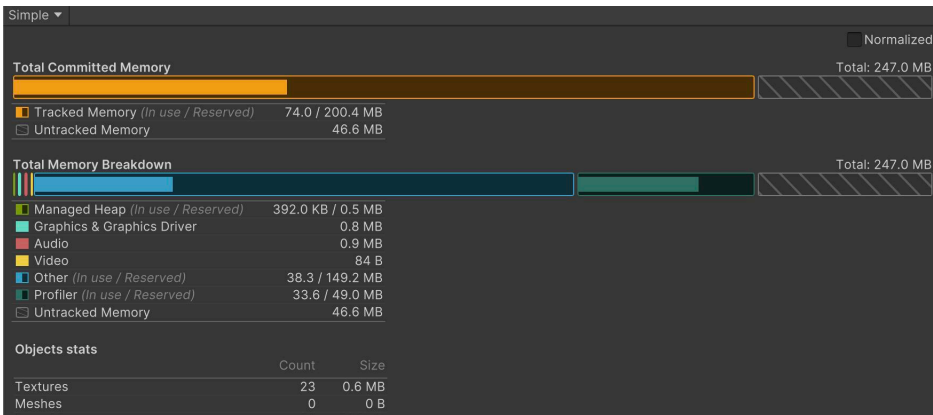
The number of times an Allocation has occurred in a frame and the total amount.

Finally, this information summarizes the use case for the Simple view.

- Understanding and monitoring heap area and Reserved expansion timing
- Checking for leaks of various assets and objects
- Monitor GC Allocation

Chapter 3 Profiling Tools

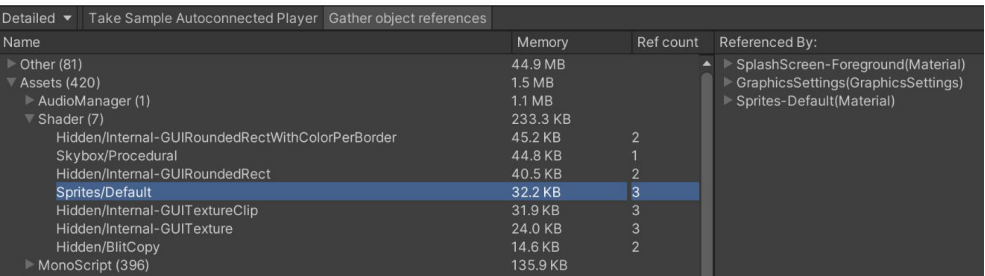
The Simple view in Unity 2021 and later has a greatly improved UI, making it easier to see the items displayed. There are no major changes in the content itself, so the knowledge introduced here can be used as is. Note, however, that some of the names have been changed. For example, GC has been renamed Managed Heap.



▲ Figure 3.14 Simple View after 2021

2. Detailed view

Detailed view looks like Figure 3.15



▲ Figure 3.15 Detailed view

The result of this view can be obtained by clicking the "Take Sample" button to take

a snapshot of the memory snapshot at that point in time. Unlike the Simple view, this view is not updated in real time, so if you want to refresh the view, you need to Take Sample again.

Figure 3.15 On the right side of the "Sample" button, there is an item called "Referenced By". This shows the objects that reference the currently selected object. If there are any assets that are leaking, the information of the object's references may help to solve the problem. This display is only shown if "Gather object references" is enabled. Enabling this feature will increase the processing time during Take Sample, but it is basically recommended to leave it enabled.

In Referenced By, you may see the notation `ManagedStaticReferences()`. This means that it is referenced by some static object. If you are familiar with the project, this information may be enough to give you some idea. If not, we recommend using "3.5 Heap Explorer".

The header items of the Detailed view are not explained here, since they mean what you see. The operation is the same as "1. Hierarchy View" in "3.1.2 CPU Usage". There is a sorting function for each header, and the items are displayed in a hierarchical view. The top node displayed in the Name item is explained here.

▼ Table 3.3 The top node of theDetailed

Name	Description
Assets	Loaded assets not included in the scene.
Not Saved	Assets generated at runtime by code. For example, objects generated by code, such as new Materiala().
Scene Memory	Assets contained in the loaded scene.
Others	Objects other than those listed above. Assignments to the various systems used by Unity.

You may not be familiar with the items listed under Others in the top node. The following is a list of items that you should know about.

System.ExecutableAndDlls

Indicates the amount of allocations used for binaries, DLLs, and so on. Depending on the platform or terminal, it may not be obtainable, in which case it is treated as 0B. The memory load for the project is not as large as the listed

values, as it may be shared with other applications using a common framework. It is better to improve Asset than to rush to reduce this item. The most effective way to do this is to reduce DLLs and unnecessary scripts. The easiest way is to change the Stripping Level. However, there is a risk of missing types and methods at runtime, so debug carefully.

SerializedFile

Indicates meta-information such as the table of objects in the AssetBundle and the Type Tree that serves as type information. This can be released by AssetBundle.Unload(true or false). Unload(false), which releases only this meta-information after the asset is loaded, is the most efficient way. Note that if the release timing and resource reference management are not done carefully, resources can be double-loaded and memory leaks can easily occur.

PersistentManager.Remapper

Remapper manages the relationship between objects in memory and on disk. Be careful not to over-expand. Specifically, if a large number of AssetBundles are loaded, the mapping area will not be sufficient and will be expanded. Therefore, it is a good idea to unload unnecessary AssetBundles to reduce the number of files loaded at the same time. Also, if a single AssetBundle contains a large number of assets that are not needed on the fly, it is a good idea to split it up.

Finally, we will summarize the cases in which the Detailed view is used based on what has been introduced so far.

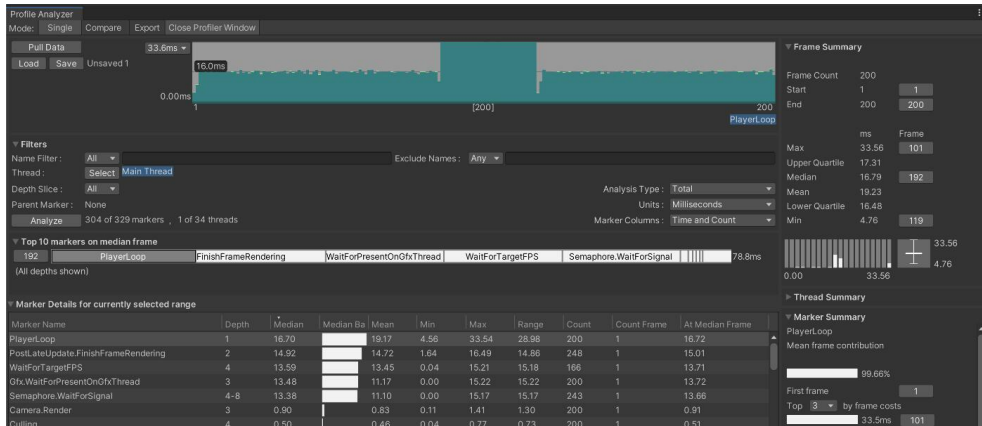
- Detailed understanding and tuning of memory at specific timing
 - Checking for unnecessary or unexpected assets
- Investigating memory leaks

3.2 Profile Analyzer

Profile Analyzer is a tool for more detailed analysis of data obtained from the Profiler's CPU Usage. While the Unity Profiler can only look at data per frame, the Profile Analyzer can obtain average, median, minimum, and maximum values based on a specified frame interval. This allows for appropriate handling of data that varies from frame to frame, making it possible to more clearly show the effects of improvement when optimization is performed. It is also a very useful tool for comparing

3.2 Profile Analyzer

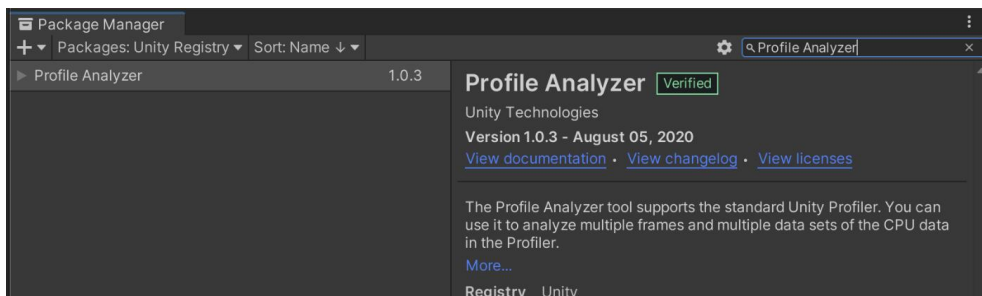
and visualizing the results of optimization because it has a function for comparing measurement data, which CPU Usage cannot do.



▲ Figure 3.16 Profile Analyzer

3.2.1 How to install

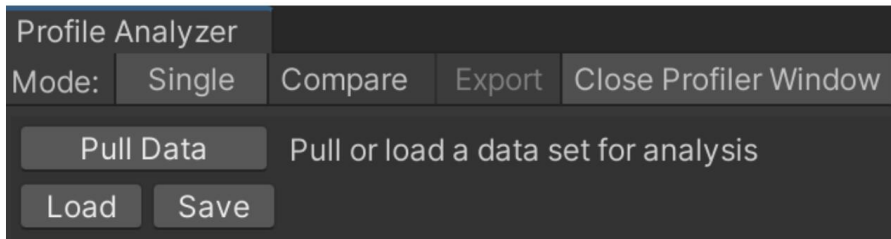
This tool can be installed from the Package Manager. Since it is officially supported by Unity, change Packages to Unity Registry and type "Profile" in the search box. After installation, you can start the tool by selecting "Window -> Analysis -> Profile Analyzer".



▲ Figure 3.17 Installation from PackageManager

3.2.2 How to operate

Profile Analyzer looks like Figure 3.18 right after startup.



▲ Figure 3.18 Immediately after startup

There are two modes of functionality: "Single" and "Compare". Single mode is used to analyze a single measurement data, while Compare mode is used to compare two measurement data.

"Pull Data" allows you to analyze data measured with the Unity Profiler and display the results. The "Pull Data" mode allows you to analyze the data measured in the Unity Profiler and display the results.

"Save" and "Load" allow you to save and load the data analyzed by Profile Analyzer. Of course, there is no problem if you keep only the Unity Profiler data. In that case, you need to load the data in Unity Profiler and do Pull Data in Profile Analyzer each time. If this procedure is troublesome, it is better to save the data as a dedicated data.

3.2.3 Analysis Result (Single mode)

The analysis result screen is structured as follows. The word "marker" appears here, but it refers to the name of the process (method name).

- Analysis section setting screen
- Screen for inputting filters for display items
- Top 10 median values of markers
- Analysis result of marker
- Frame Summary

- Thread Summary
- Summary of selected markers

Let's take a look at each of these display screens.

1. analysis section setting screen

The processing time for each frame is displayed, with all frames initially selected. The frame interval can be changed by dragging like Figure 3.19, so adjust it if necessary.



▲ Figure 3.19 Specifying a frame interval

2. filter input screen

The filter input screen allows filtering of analysis results.

▲ Figure 3.20 Filter Input Screen

Each item is as follows.

Chapter 3 Profiling Tools

▼ Table 3.4 Items of Filters

Item Name	Description
Name Filter	Filter by the name of the process you want to search.
Exclude Filter	Filter by the name of the process you want to exclude from the search.
Thread	The selected threads will be displayed in the analysis results. If you need information on other threads, add them.
Depth Slice	This is the number of slice in the Hierarchy in CPU Usage. For example, if Depth is 3, the third hierarchy is displayed.
Analysis Type	Total and Self can be switched. This is the same as the header item introduced in CPU Usage.
Units	Time display can be changed to milliseconds or microseconds.
Marker Columns	Change the header display of analysis results.

When Depth Slice is set to All, the top node called PlayerLoop is displayed, or different layers of the same process are displayed, which can be difficult to see. In such cases, it is recommended to fix Depth to 2~3 and set it so that subsystems such as rendering, animation, and physics are displayed.

3) Top 10 Median Marker Values

This screen shows only the top 10 markers sorted by the median processing time for each marker. You can see at a glance how much processing time each of the top 10 markers occupies.



▲ Figure 3.21 Median of Top 10 markers

4. analysis results of markers

The analysis results of each marker are displayed. It is a good idea to analyze the process that should be improved based on the process name listed in Marker Name and the values of Median and Mean. If you move the mouse pointer over a header item, a description of the item will be displayed, so please refer to it if you do not understand the content.

▼ Marker Details for currently selected range

Marker Name	Depth	Media	Media	Mean	Min	Max	Range	Count	Count Fr	At Median F
PlayerLoop	1	16.71		19.24	4.56	33.54	28.98	199	1	16.71
PostLateUpdate.FinishFrameRendering	2	14.93		14.80	1.73	16.49	14.76	247	1	14.83
WaitForTargetFPS	4	13.59		13.53	0.04	15.21	15.18	165	1	13.35
Gfx.WaitForPresentOnGfxThread	3	13.49		11.22	0.00	15.22	15.22	199	1	13.35
Semaphore.WaitForSignal	4-8	13.38		11.16	0.00	15.17	15.17	242	1	13.28
Camera.Render	3	0.90		0.82	0.11	1.41	1.30	199	1	1.16
Culling	4	0.50		0.46	0.04	0.77	0.73	199	1	0.69
SceneCulling	5	0.35		0.33	0.02	0.62	0.60	199	1	0.52
PostLateUpdate.ProfilerEndFrame	2-3	0.30		0.30	0.13	1.10	0.97	247	1	0.33
Profiler.FlushCounters	3	0.28		0.26	0.03	1.10	1.07	199	1	0.33
PrepareSceneNodes	6	0.27		0.25	0.01	0.53	0.52	199	1	0.43
Profiler.FlushMemoryCounters	4	0.22		0.20	0.02	1.08	1.06	199	1	0.23

▲ Figure 3.22 Analysis results for each process

Mean and Median

The mean is the value obtained by adding all values together and dividing by the number of data. The median, on the other hand, is the value that lies in the middle of the sorted data. In the case of an even number of data, the average value is taken from the data before and after the median.

The average has the property that it is susceptible to data with values that are extremely far apart. If there are frequent spikes or the sampling number is not sufficient, it may be better to refer to the median.

Figure 3.23 is an example of a large difference between the median and the mean.

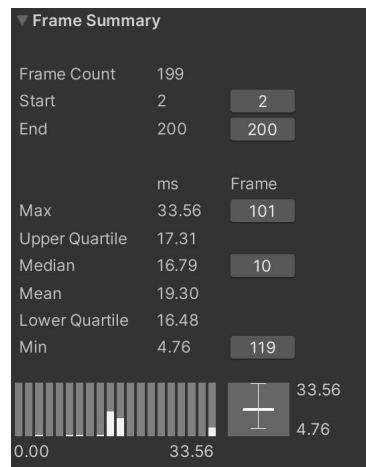
Time (ms)		Average (ms)	Median (ms)
27	28		
29	30	557.833...	
33	3200		29.5

▲ Figure 3.23 Median and Mean

Analyze your data after knowing the characteristics of these two values.

5. frame summary

This screen shows the frame statistics of the measured data.



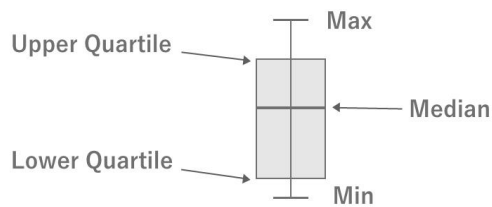
▲ Figure 3.24 Frame Summary Screen

This screen displays interval information for the frame being analyzed and the degree of variation in the values using a boxplot or histogram. Box plots require an understanding of quartiles. Quartiles are defined values with the data sorted as Table 3.5.

▼ Table 3.5 Quartiles

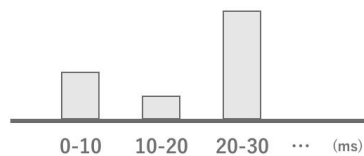
Name	Description
Minimum value (Min)	Minimum value
Lower Quartile	Value located 25% of the way from the minimum
Median	The value located at the 50% position from the minimum
Upper Quartile	Value in the 75th percentile from the minimum
Maximum Value (Max)	Maximum value

The interval between 25% and 75% is boxed, which is called a box-and-whisker graph.



▲ Figure 3.25 Box-and-whisker graph

The histogram shows processing time on the horizontal axis and the number of data on the vertical axis, which is also useful for viewing data distribution. In the frame summary, you can check the interval and the number of frames by hovering the cursor over them.

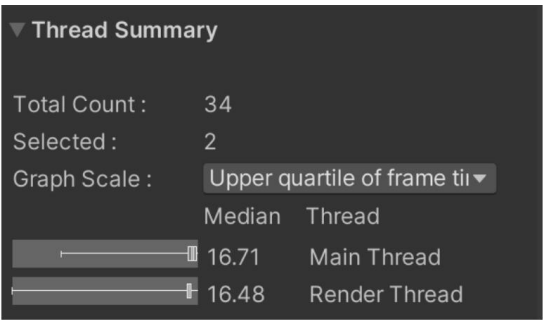


▲ Figure 3.26 Histograms

After understanding how to see these diagrams, it is a good idea to analyze the features.

6. thread summary

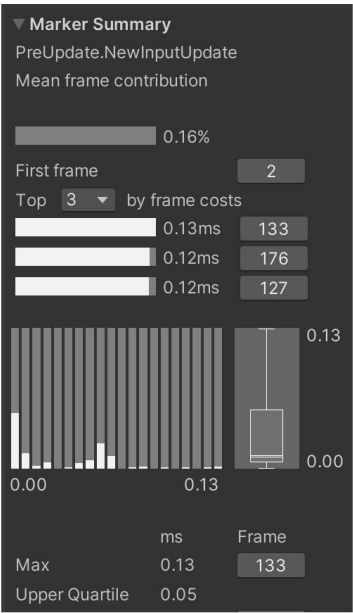
This screen shows statistics for the selected thread. You can see a box-and-whisker diagram for each thread.



▲ Figure 3.27 Frame Summary Screen

7. summary of the currently selected marker

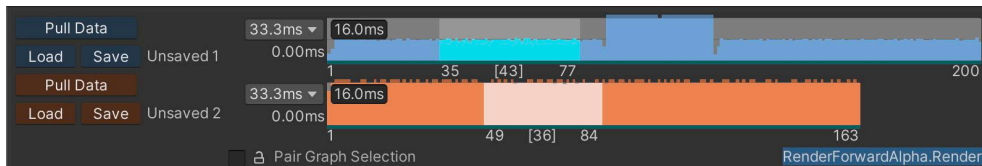
This is a summary of the marker selected on the "4. analysis results of markers" screen. The processing time for the currently selected marker is shown in a box-and-whisker diagram or histogram.



▲ Figure 3.28 Summary of selected markers

3.2.4 Analysis results (Compare mode)

In this mode, two sets of data can be compared. The interval to be analyzed can be set for each of the upper and lower data.



▲ Figure 3.29 Comparison data settings

The usage of the screen is almost the same as Single mode, but the words "Left" and "Right" appear in various screens like Figure 3.30.

▼ Marker Comparison for currently selected range							
Marker Name	Left Median	<	>	Right Median	Diff	Abs Diff	Count
Gfx.WaitForGfxCommandsFromMainThread	0.66			31.85	31.19	31.19	76
WaitForTargetFPS	13.90			32.44	18.55	18.55	43
Semaphore.WaitForSignal	14.64			31.85	17.21	17.21	121
PlayerLoop	16.70			33.29	16.59	16.59	43
PostLateUpdate.FinishFrameRendering	15.18			0.40	-14.78	14.78	43
Gfx.PresentFrame	14.29			0.10	-14.20	14.20	43

▲ Figure 3.30 Comparison of markers

This shows which data is which, and matches the color shown at Figure 3.29. Left is the top data and Right is the bottom data. This mode will make it easier to analyze whether the tuning results are good or bad.

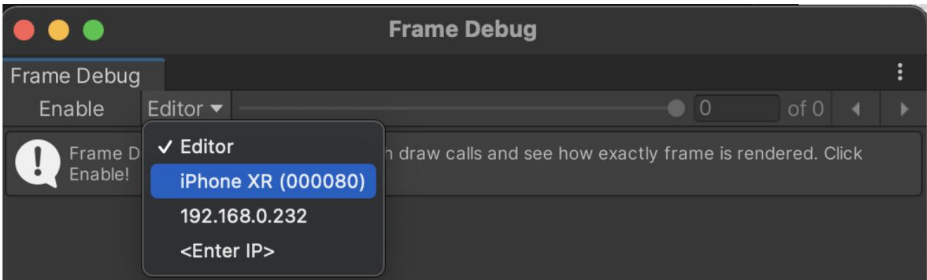
3.3 Frame Debugger

The Frame Debugger is a tool that allows you to analyze how the currently displayed screen is rendered. This tool is installed by default in the editor and can be opened by selecting "Window -> Analysis -> Frame Debugger".

It can be used in the editor or on the actual device. When using it on an actual device, a binary built with "Development Build" is required, as is the Unity Profiler.

Chapter 3 Profiling Tools

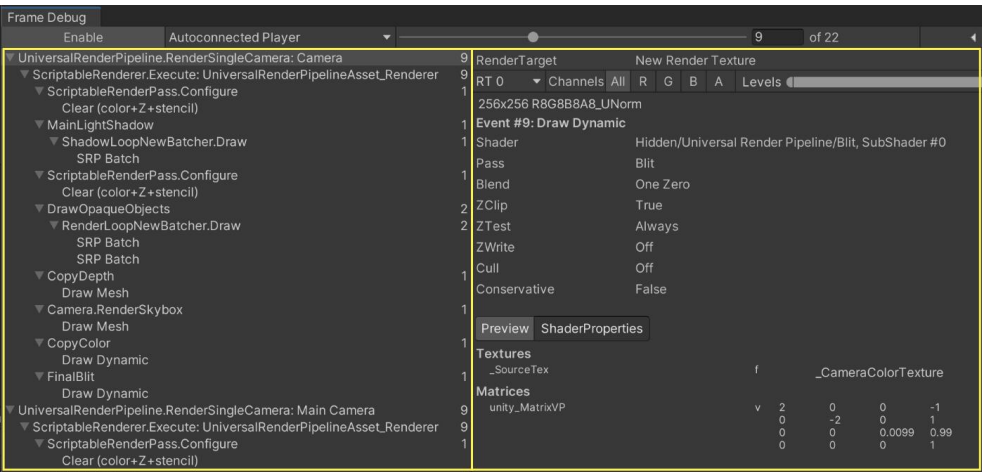
Start the application, select the device connection, and press "Enable" to display the drawing instruction.



▲ Figure 3.31 FrameDebugger connection screen

3.3.1 Analysis screen

Click "Enable" to display the following screen.



▲ Figure 3.32 FrameDebugger Capture

The left frame shows a single drawing instruction per item, with the instructions issued in order from top to bottom. The right frame shows detailed information about drawing instructions. You can see which Shader was processed with what properties.

While looking at this screen, analyze with the following in mind.

- Are there any unnecessary instructions?
- Whether drawing batching is working properly or not
- Is the resolution of the drawing target too high?
- Is an unintended Shader being used?

3.3.2 Detailed Screen

The contents of the right frame of Figure 3.32 introduced in the previous section are explained in detail.

Operation Panel

First, let's look at the operation panel in the upper section.

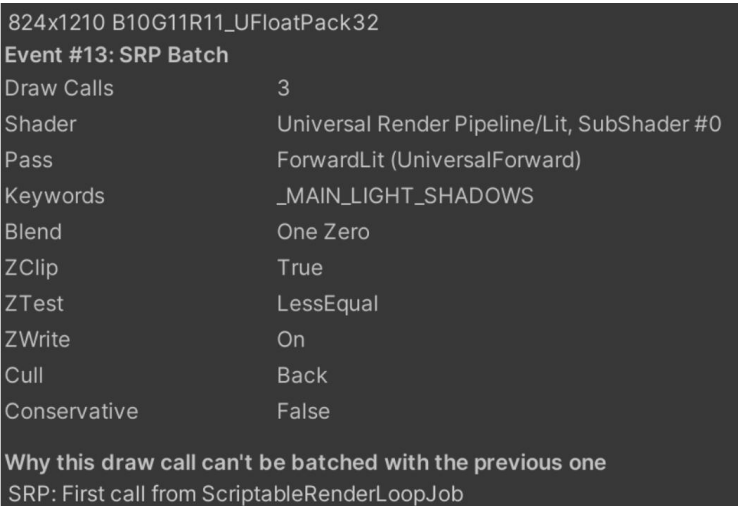


▲ Figure 3.33 Operation panel in the upper section

The part marked "RT0" can be changed when there are multiple render targets. This is especially useful when using multiple render targets to check the rendering status of each target. Channels can be changed to display all RGBA or only one of the channels. Levels is a slider that allows you to adjust the brightness of the resulting rendering. This is useful, for example, to adjust the brightness of a dark rendering, such as ambient or indirect lighting, to make it easier to see.

Drawing Overview

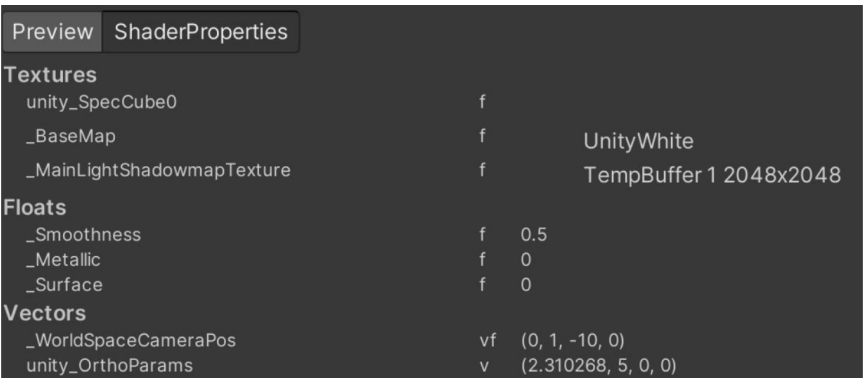
This area provides information on the resolution and format of the destination. Obviously, you will be able to notice immediately if there is a drawing destination with a higher resolution. Other information such as the Shader name used, Pass settings such as Cull, and keywords used can also be found. The sentence "Why this~" listed at the bottom describes why the drawing could not be batching. Figure 3.34 In the case of "Why this~," it states that the first drawing call was selected and therefore batching was not possible. Since the causes are described in such detail, you can rely on this information to make adjustments if you want to devise batching.



▲ Figure 3.34 Overview of the middle drawing

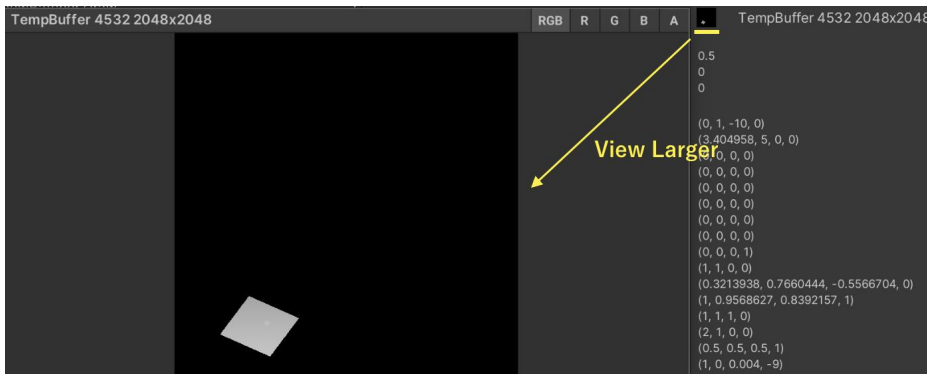
Detailed information on the Shader property

This area describes the property information of the Shader being drawn. This is useful for debugging.



▲ Figure 3.35 Detailed information on Shader properties in the lower row

Sometimes it is necessary to check in detail the state of Texture2D displayed in the property information. To do so, click on the image while holding down the Command key on a Mac (Control key on Windows) to enlarge the image.



▲ Figure 3.36 Enlarge Texture2D preview

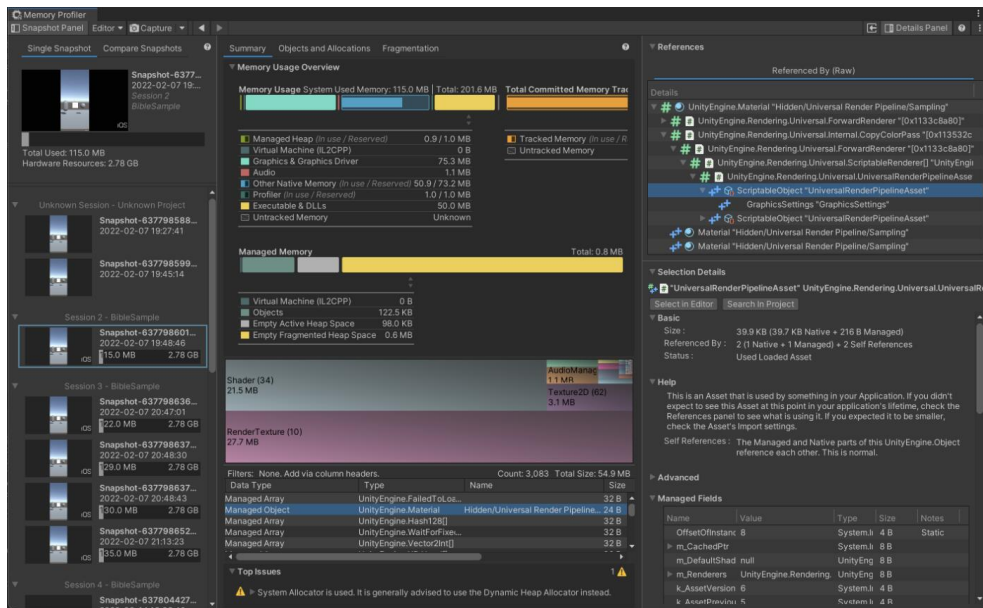
3.4 Memory Profiler

Memory Profiler is an official tool provided by Unity as a Preview Package. Compared to the Memory module of the Unity Profiler, it is superior in the following main points.

- Captured data is saved locally with screenshots
- The amount of memory occupied by each category is visualized and easy to understand
- Data can be compared

The UI of the Memory Profiler has changed significantly between v0.4 and later versions. This book uses v0.5, which is the latest version at the time of writing. For v0.4 or later versions, Unity 2020.3.12f1 or later version is required to use all features. In addition, v0.4 and v0.5 look the same at first glance, but v0.5 has been significantly updated. In particular, object references are now much easier to follow, so we basically recommend using v0.5 or later.

Chapter 3 Profiling Tools

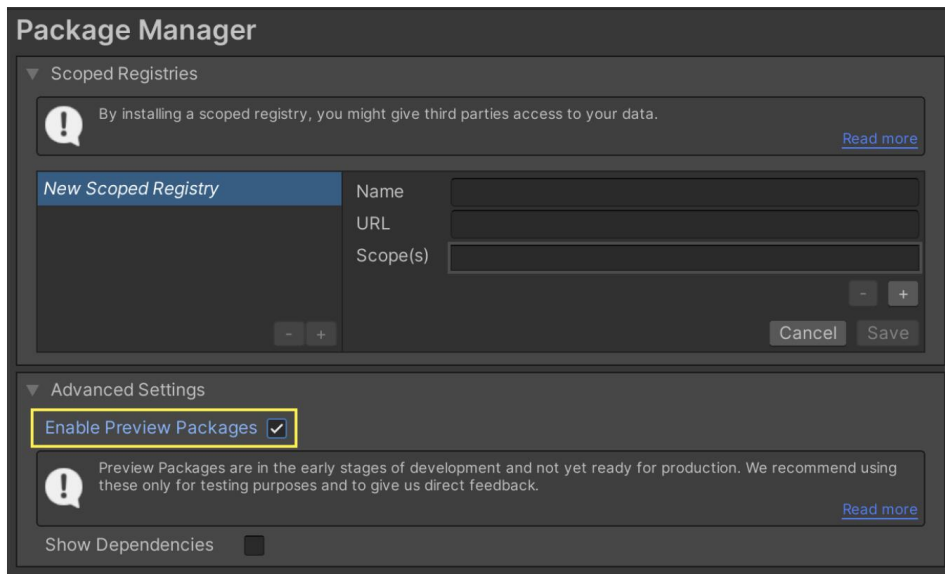


▲ Figure 3.37 Memory Profiler

3.4.1 How to install

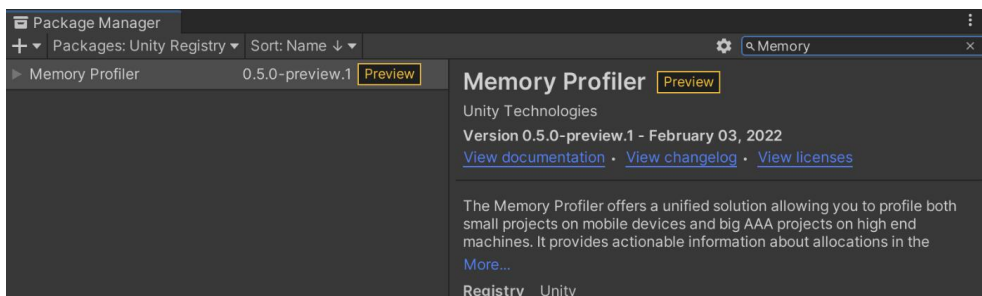
In Unity 2020, you need to enable "Enable Preview Packages" in "Project Settings -> Package Manager" for the Preview version packages.

3.4 Memory Profiler



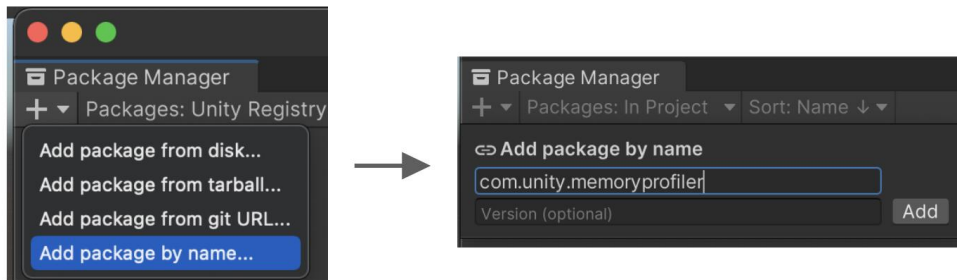
▲ Figure 3.38 Enable Preview Packages

Then install the Memory Profiler from Package in the Unity Registry. After installation, go to "Window -> Analysis -> Memory Profiler" to launch the tool.



▲ Figure 3.39 Install from PackageManager

In Unity 2021 and later, the method of adding packages has been changed. To add a package, click on "Add Package by Name" and enter "com.unity.memoryprofiler".



▲ Figure 3.40 How to add after 2021

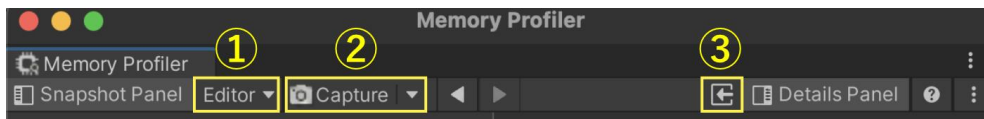
3.4.2 How to Operate

Memory Profiler consists of four major components.

- Toolbar
- Snapshot Panel
- Measurement Results
- Detail Panel

Explanations are given for each area.

1. tool bar



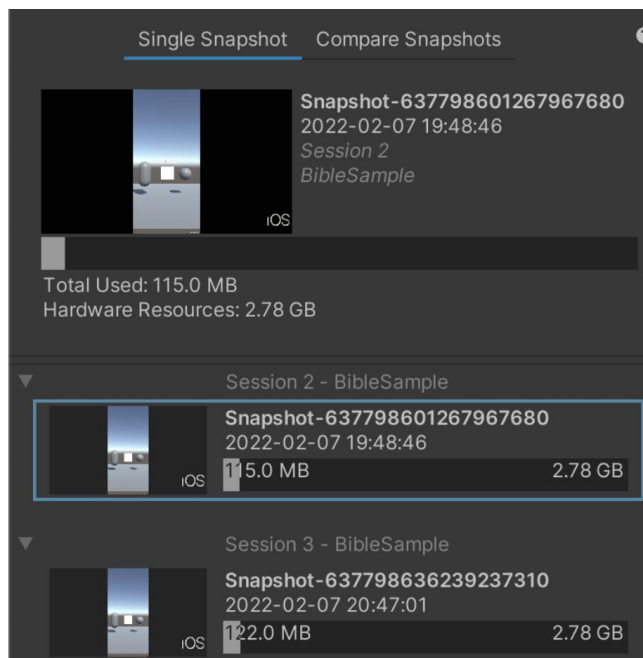
▲ Figure 3.41 Toolbar area

Figure 3.41 indicates a capture of the Header. The button ① allows you to select the measurement target. The button (2) measures the memory at the time when it is pressed. Optionally, you can choose to measure only Native Objects or disable screenshots. The basic default settings should be fine. Clicking the button (③) will load the measured data. Clicking the "Snapshot Panel" or "Detail Panel" button will

show or hide the information panels on the left and right sides of the screen. If you only want to see the tree map, it is better to hide them. You can also click the "?" to open the official document.

There is one important point to note regarding the measurement. One thing to note about measurement is that the memory required for measurement is newly allocated and will not be released again. However, it does not increase infinitely and will eventually settle down after several measurements. The amount of memory allocated at measurement time will depend on the complexity of the project. If you do not know this assumption, be careful because you may mistakenly think there is a leak when you see the amount of memory usage ballooning.

Snapshot Panel



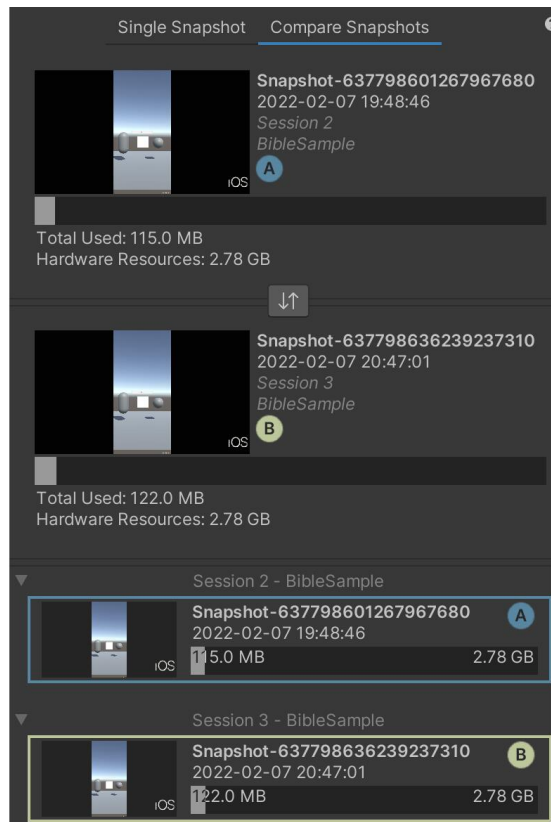
▲ Figure 3.42 Snapshot Panel (Single)

The Snapshot Panel displays the measured data and allows you to choose which data to view. The data is organized by session, from the time the application is launched to the time it is terminated. You can also delete or rename the measured

Chapter 3 Profiling Tools

data by right-clicking on it.

"Single Snapshot" and "Compare Snapshots" are available at the top. Clicking "Compare Snapshots" changes the display to a UI for comparing measurement data.



▲ Figure 3.43 Snapshot Panel (Compare)

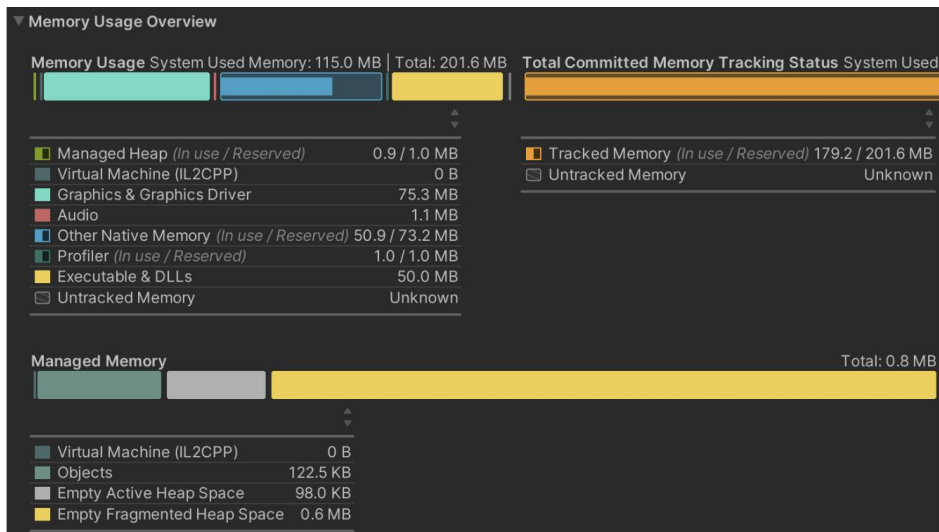
"A" is the data selected in Single Snapshot and "B" is the data selected in Compare Snapshots. By clicking on the "Replace" button, "A" and "B" can be switched without returning to the Single Snapshot screen.

3. measurement results

There are three tabs for measurement results: "Summary," "Objects and Allocations," and "Fragmentation." This section describes Summary, which is frequently used, and briefly describes the other tabs as supplementary information. The upper part of

3.4 Memory Profiler

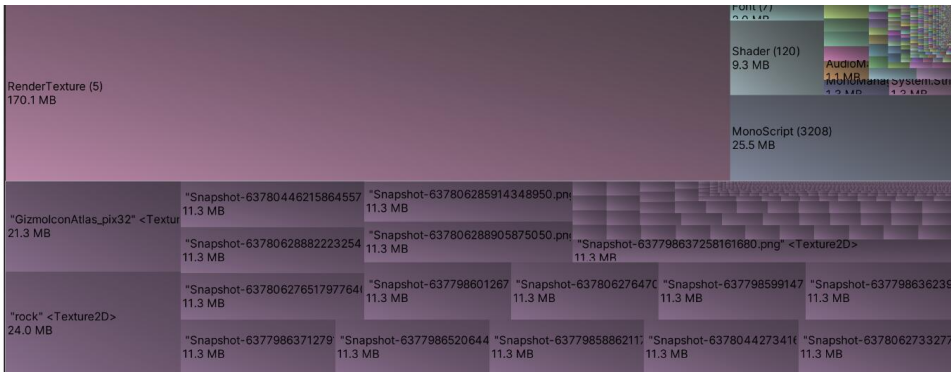
the Summary screen is an area called Memory Usage Overview, which displays an overview of the current memory. Clicking on an item displays an explanation in the Detail Panel, so it is a good idea to check items you do not understand.



▲ Figure 3.44 Memory Usage Overview

The next area of the screen is called the Tree Map, which graphically displays memory usage for each category of objects. By selecting each category, you can check the objects within the category. Figure 3.45 In the following example, the Texture2D category is selected.

Chapter 3 Profiling Tools



▲ Figure 3.45 Tree Map

The bottom part of the screen is called Tree Map Table. Here, the list of objects is arranged in a table format. The displayed items can be grouped, sorted, and filtered by pressing the header of the Tree Map Table.



▲ Figure 3.46 Header Operations

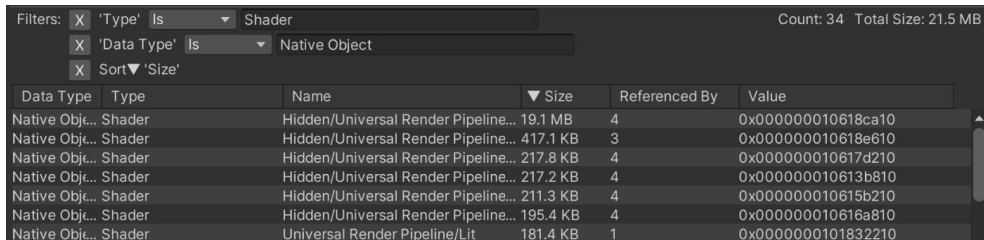
Especially, grouping the Types makes it easier to analyze, so please use it proactively.

Data Type	[Type]	Name	Size	Referenced By
	▶ AudioListener (2)		432 B	2
Native Obj...	AudioManager (1)	AudioManager	1.1 MB	0
Native Obj...	BoxCollider (1)	Cube	256 B	1
Native Obj...	BuildSettings (1)	BuildSettings	0.6 KB	0
	▶ Camera (2)		8.5 KB	4

▲ Figure 3.47 Grouping by Type

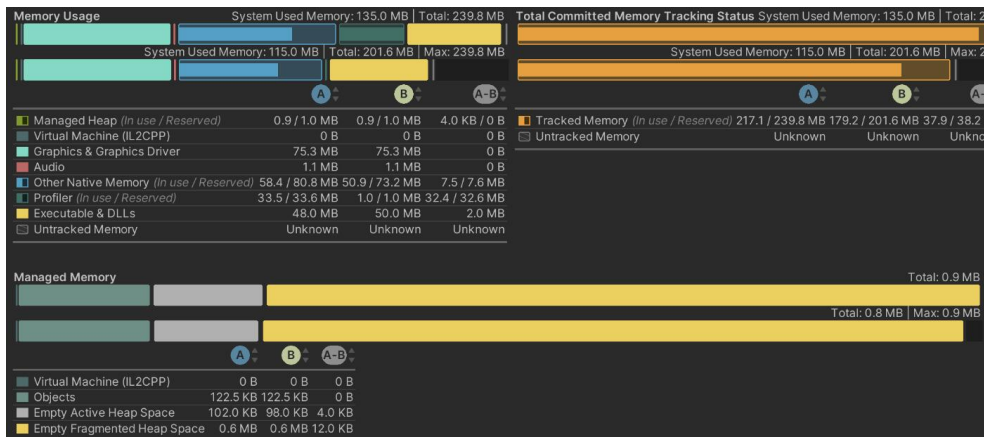
3.4 Memory Profiler

When a category is selected in the Tree Map, a filter is automatically set to display only objects in that category.



▲ Figure 3.48 Automatic Filter Settings

Finally, the UI changes when Compare Snapshots is used. Memory Usage Overview displays the differences for each object.



▲ Figure 3.49 Memory Usage Overview in Compare Snapshots

In the Tree Map Table, a Diff item is added to the Header. Diffs can be of the following types

▼ Table 3.6 Tree Map Table (Compare)

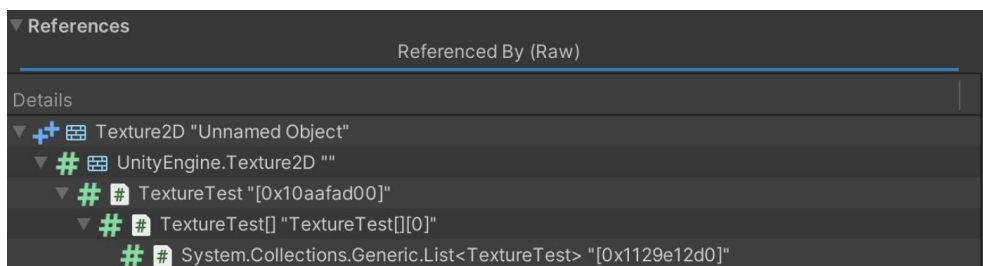
Diff	Description
Same	A, B same object
Not in A (Deleted)	Object in A but not in B
Not in B (New)	Object not in A but in B

Chapter 3 Profiling Tools

By looking at this information, it is possible to check whether memory is increasing or decreasing.

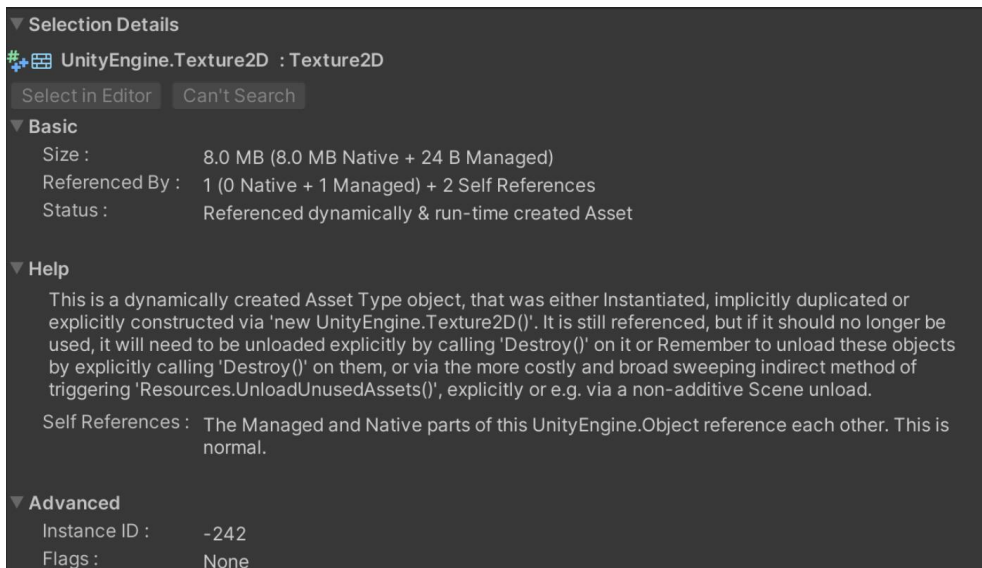
Detail Panel

This panel is used when you want to track the reference relationship of the selected object. By checking this Referenced By, you will be able to figure out what is causing the continued reference grabbing.



▲ Figure 3.50 Referenced By

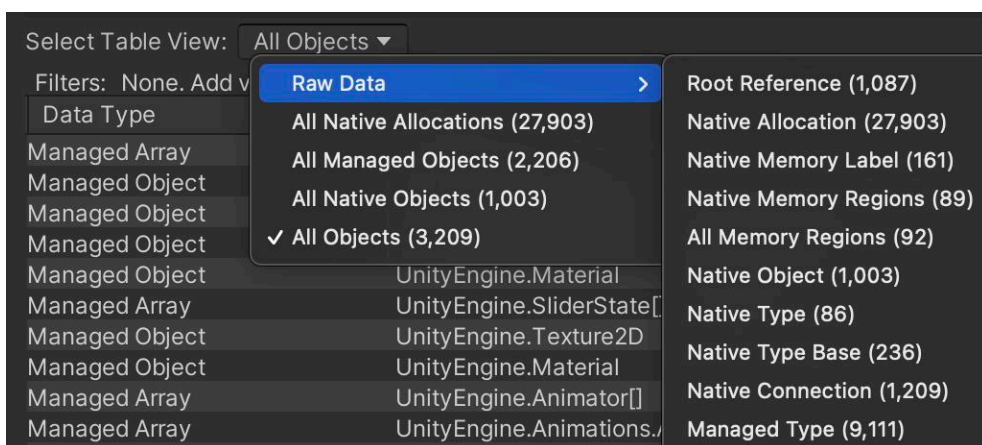
The bottom section, Selection Details, contains detailed information about the object. Among them, the "Help" section contains advice on how to release it. You may want to read it if you are not sure what to do.



▲ Figure 3.51 Selection Details

Supplemental: Measurement results other than Summary

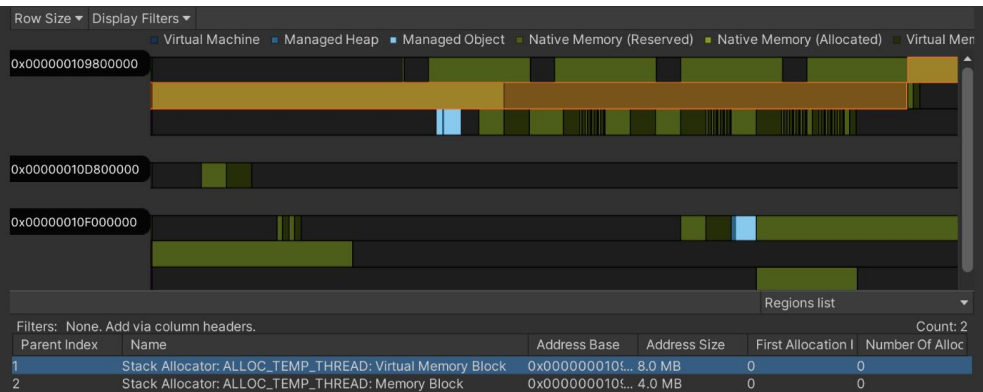
"Objects and Allocations" differs from Summary in that more detailed information such as allocations can be viewed in table format.



▲ Figure 3.52 Table View Specifications

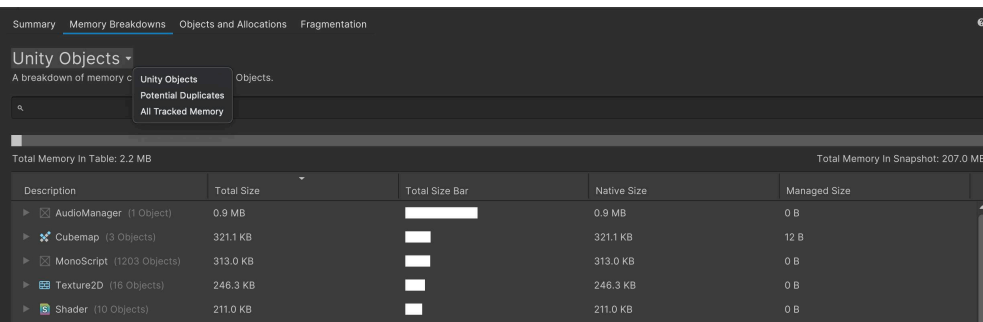
Chapter 3 Profiling Tools

"Fragmentation" visualizes the virtual memory status and can be used to investigate fragmentation. However, it may be difficult to use because it contains a lot of non-intuitive information such as memory addresses.



▲ Figure 3.53 Fragmentation

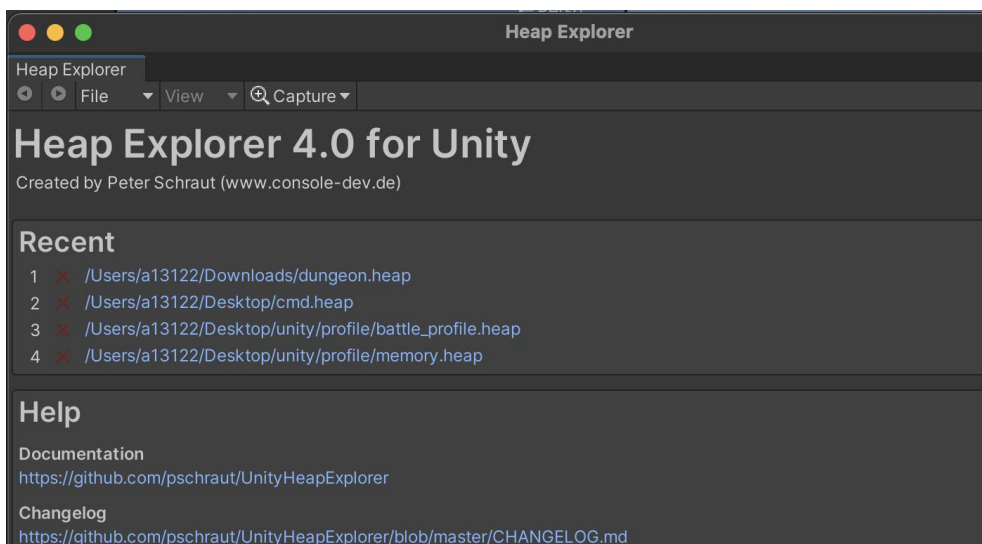
A new feature called "Memory Breakdowns" has been added since v0.6 of Memory Profiler. Unity 2022.1 or later is required, but it is now possible to view TreeMaps in list view and object information such as Unity Subsystems. Other new features include the ability to check for possible duplicate objects.



▲ Figure 3.54 Memory Breakdowns

3.5 Heap Explorer

Heap Explorer is an open source tool from private developer Peter77^{*1}. Like Memory Profiler, this tool is often used to investigate memory. Memory Profiler was very labor intensive to track down references in versions prior to 0.4 because they were not displayed in a list format. Although this has been improved in 0.5 and later, there may be some who use a version of Unity that is not supported. It is still very valuable as an alternative tool in such cases, so we would like to cover it in this issue.



▲ Figure 3.55 Heap Explorer

3.5.1 How to install

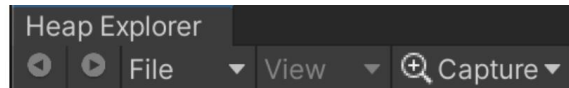
Copy the Package URL's listed in the GitHub repository^{*2} and add it from the Add Package from Git url in the Package Manager. After installation, you can launch the tool from "Window -> Analysis -> Memory Profiler".

^{*1} <https://github.com/pschraut>

^{*2} <https://github.com/pschraut/UnityHeapExplorer>

3.5.2 How to use

The toolbar of Heap Explorer looks like this



▲ Figure 3.56 Heap Explorer toolbar

Left and right arrows

Allows you to go back and forward in an operation. This is especially useful for tracking references.

File

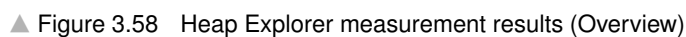
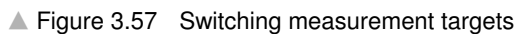
Allows saving and loading of measurement files. The file is saved with a .heap extension.

View

Switch between different display screens. There are various types, so if you are interested, please refer to the documentation.

Capture

Capture measurement. However, the **measurement target cannot be changed in Heap Explorer**. The target must be changed in the Unity Profiler or other tools provided by Unity. Save saves the measurement to a file and displays the results, while Analyze displays the results without saving. Note that, as with the Memory Profiler, memory allocated during measurement is not released.



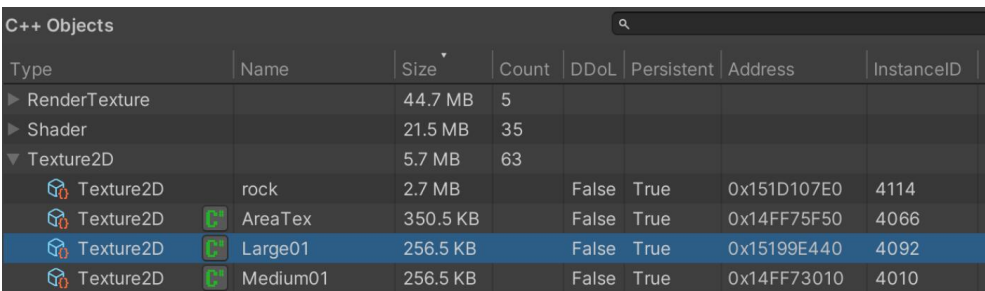
Chapter 3 Profiling Tools








and Managed Memory Usage, which are indicated by green lines. Click the "Investigate" button to see the details of each category.

In the following sections, we will focus on the important parts of the category details.

1. object

When Native Memory is Investigate, C++ Objects are displayed in this area. In case of Managed Memory, C# Objects will be displayed in this area.



Type	Name	Size	Count	DDoL	Persistent	Address	InstanceID
▶ RenderTexture		44.7 MB	5				
▶ Shader		21.5 MB	35				
▼ Texture2D		5.7 MB	63				
 Texture2D	rock	2.7 MB		False	True	0x151D107E0	4114
 Texture2D	 AreaTex	350.5 KB		False	True	0x14FF75F50	4066
 Texture2D	 Large01	256.5 KB		False	True	0x15199E440	4092
 Texture2D	 Medium01	256.5 KB		False	True	0x14FF73010	4010

▲ Figure 3.59 Object display area

There are some unfamiliar items in the header.

DDoL

DDoL stands for "Don't Destroy On Load. You can see if the object is designated as an object that will not be destroyed after a scene transition.







Persistent

Indicates whether the object is a persistent object or not. This is an object that is automatically created by Unity at startup.

The display area introduced below will be updated by selecting the object Figure 3.59.

Referenced by




The object from which the target object is referenced is displayed.

Referenced by 2 object(s)			
Type	C++ Name	Address	
 PostProcessD  	PostProcessData	0x11413E660	
 ...Texture2D  	Medium06	0x1112B97C0	

▲ Figure 3.60 Referenced by

References to
































Displays objects that are referenced by the target object.

References to 1 object(s)			
Type	C++ Name	Address	
 GCHandle  	UnityEngine.Texture2D	0x1112B97C0	

▲ Figure 3.61 References to

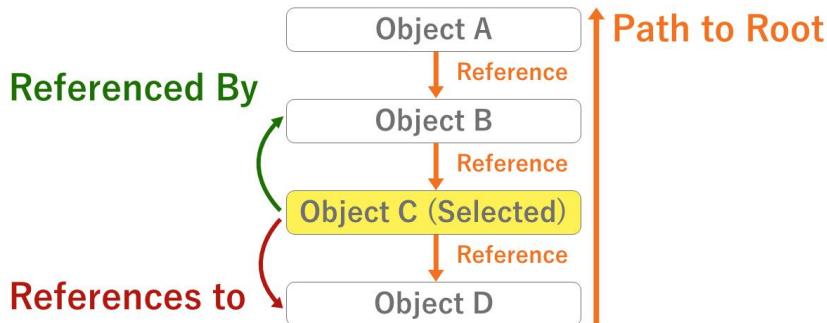
Path to Root

Displays the root objects that reference the target object. This is useful when investigating memory leaks, as it allows you to see what is holding the reference.

2 Path(s) to Root				
Type	C++ Name	Depth	Address	
▼   UnityEngine.Rendering.RenderPipelineManager	static s_CurrentPipelineAsset	6		
 ...Universal.UniversalRenderPipelineAsset  	UniversalRenderPipelineAsset		0x106F1B	
 UniversalRenderPipelineAsset	  UniversalRenderPipelineAsset		0x1519CF	
 ForwardRendererData	  UniversalRenderPipelineAsset_Renderer		0x1519D2	
 PostProcessData	  PostProcessData		0x1519D2	
 Texture2D	  Large01		0x15199E	
▼   GraphicsSettings	GraphicsSettings	5	0x14FF50	
 UniversalRenderPipelineAsset	  UniversalRenderPipelineAsset		0x1519CF	
 ForwardRendererData	  UniversalRenderPipelineAsset_Renderer		0x1519D2	
 PostProcessData	  PostProcessData		0x1519D2	
 Texture2D	  Large01		0x15199E	

▲ Figure 3.62 Path to Root

The following image summarizes the previous items.



▲ Figure 3.63 Reference Image

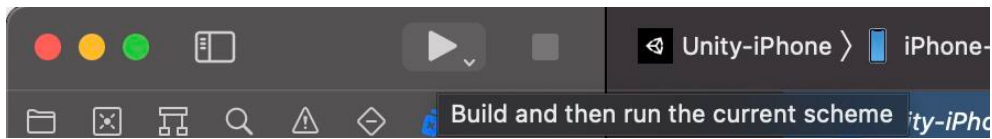
As introduced so far, Heap Explorer provides a complete set of functions necessary for investigating memory leaks and memory. It is also very lightweight, so please consider using this tool. If you like it, it would be better if you add a Star as a token of your appreciation.

3.6 Xcode

Xcode is an integrated development environment tool provided by Apple. When you set the target platform as iOS in Unity, the build result will be an Xcode project. It is recommended to use Xcode for rigorous verification, as it provides more accurate values than Unity. In this section, we will touch on three profiling tools: Debug Navigator, GPU Frame Capture, and Memory Graph.

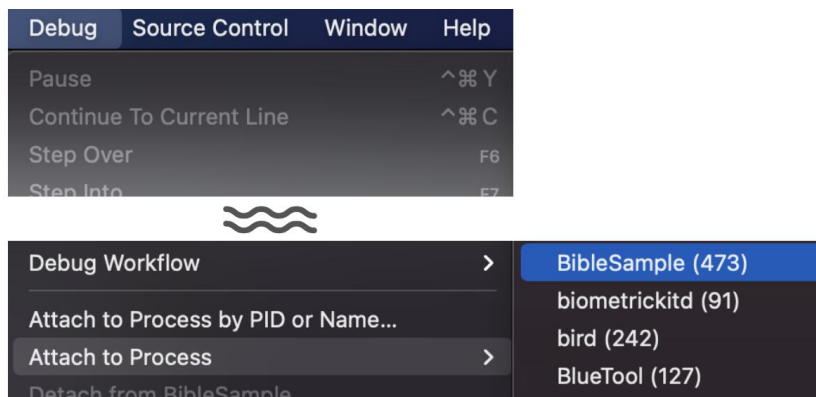
3.6.1 Profiling Methods

There are two ways to profile from Xcode. The first is to build and run the application directly from Xcode. Figure 3.64 The first method is to build the application directly from Xcode and run it on the terminal. Settings such as certificates when performing a build are omitted from this document.



▲ Figure 3.64 Xcode's Execute button

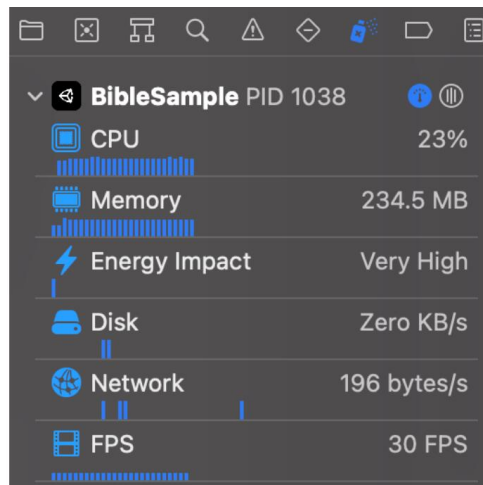
The second method is to attach the running application to the Xcode debugger. This can be profiled by selecting the running process from "Debug -> Attach to Process" in the Xcode menu after running the application. However, the certificate at build time must be for developer (Apple Development). Note that Ad Hoc or Enterprise certificates cannot be used to attach.



▲ Figure 3.65 Debugger Attach in Xcode

3.6.2 Debug Navigator

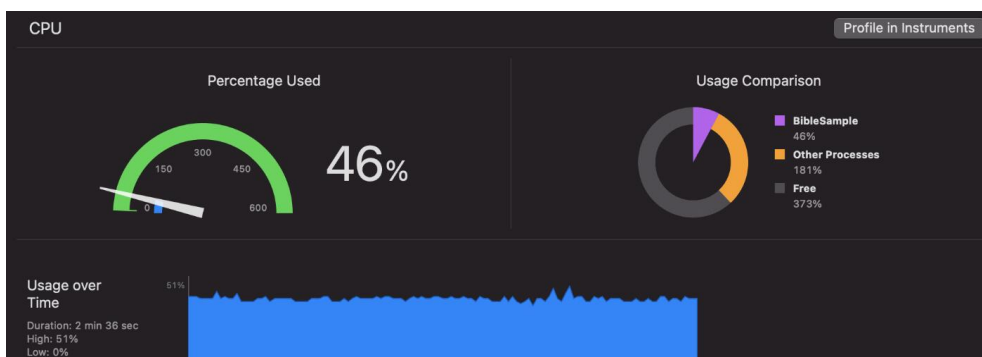
Debug Navigator allows you to check debugging gauges such as CPU and Memory just by running the application from Xcode. Six items are displayed by pressing the spray mark of Figure 3.66 after running the application. Alternatively, you can open it from "View -> Navigators -> Debug" in the Xcode menu. Each item will be explained in the following sections.



▲ Figure 3.66 Selecting Debug Navigator

1. CPU Gauge

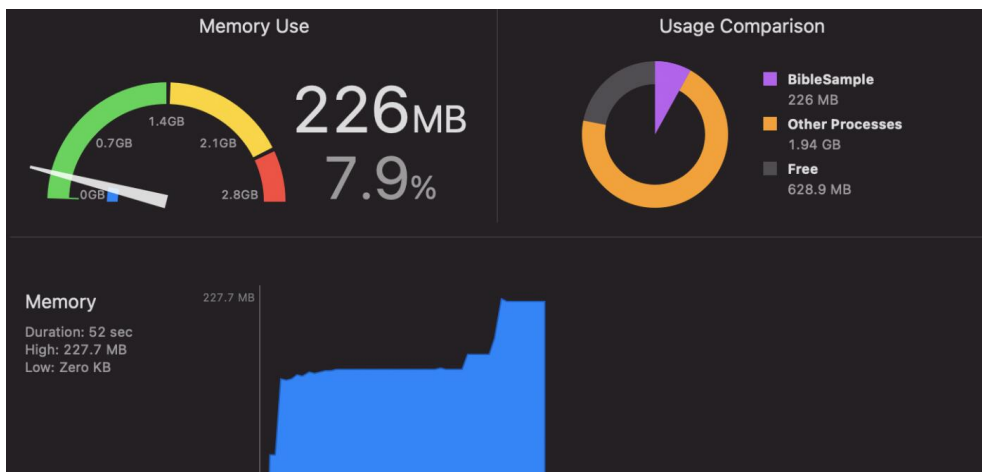
You can see how much CPU is being used. You can also see the usage rate of each thread.



▲ Figure 3.67 CPU Gauge

2. Memory Gauge

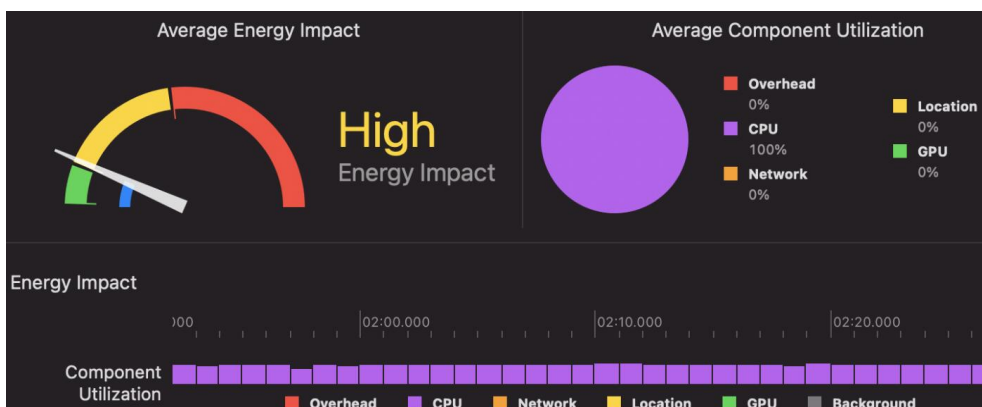
An overview of memory consumption can be viewed. Detailed analysis such as breakdown is not available.



▲ Figure 3.68 Memory Gauge

Energy Gauge

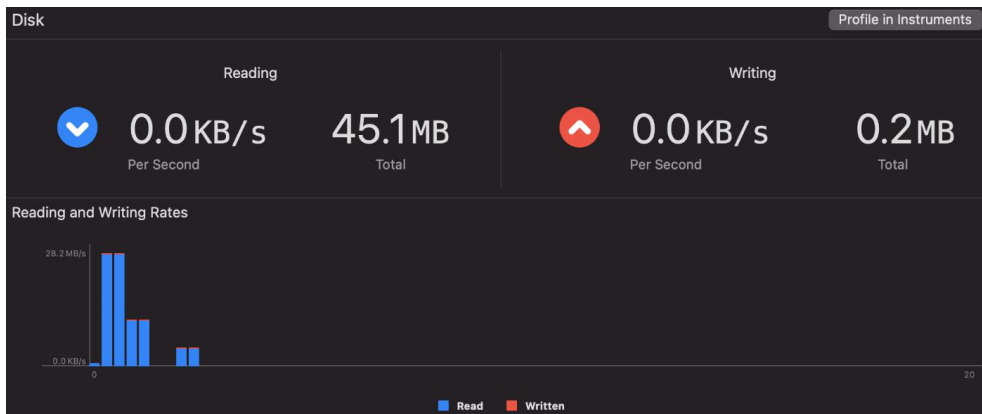
This gauge provides an overview of power consumption. You can get a breakdown of CPU, GPU, Network, etc. usage.



▲ Figure 3.69 Energy Gauge

Disk Gauge

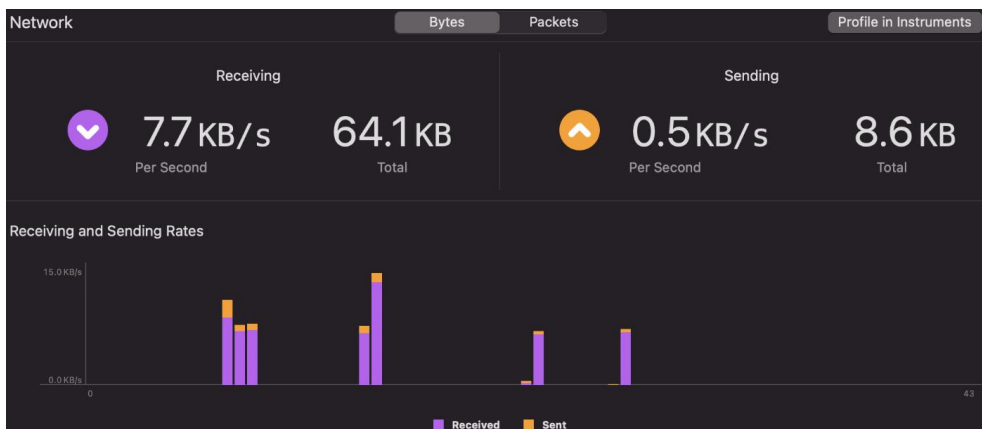
This gauge provides an overview of File I/O. It will be useful to check if files are being read or written at unexpected times.



▲ Figure 3.70 Disk Gauge

5. Network Gauge

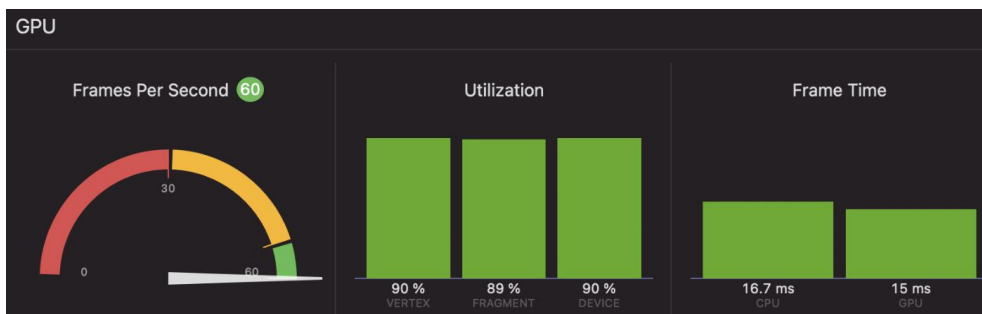
This gauge provides an overview of network communication. Like the Disk gauge, it is useful for checking for unexpected communication.



▲ Figure 3.71 Network Gauge

FPS Gauge

This gauge is not displayed by default. It is displayed when GPU Frame Capture, described at "3.6.3 GPU Frame Capture", is enabled. You can check not only the FPS, but also the utilization of shader stages and the processing time of each CPU and GPU.



▲ Figure 3.72 FPS Gauge

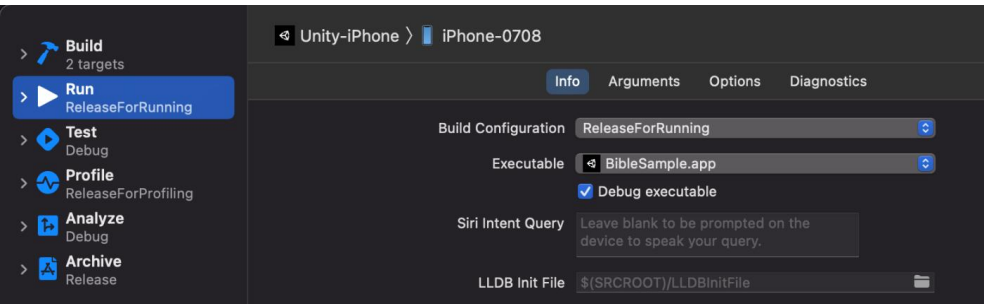
3.6.3 GPU Frame Capture

GPU Frame Capture is a tool that allows frame debugging on Xcode. Similar to Unity's Frame Debugger, you can check the process until rendering is completed. Compared to Unity, there is more information at each shader stage, so it may be useful for analyzing and improving bottlenecks. The following is an explanation of how to use it.

1. preparation

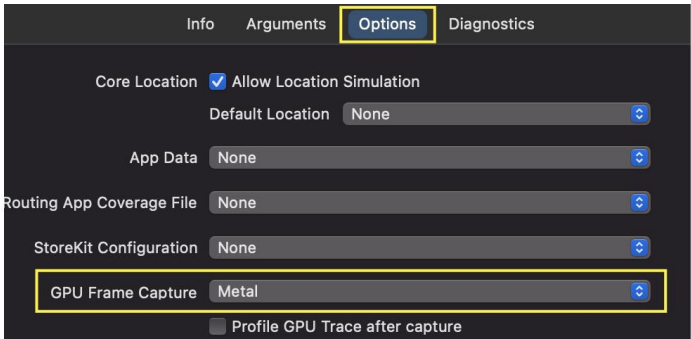
To enable GPU Frame Capture in Xcode, you need to edit the scheme. First, open the scheme edit screen by selecting "Product -> Scheme -> Edit Scheme."

Chapter 3 Profiling Tools



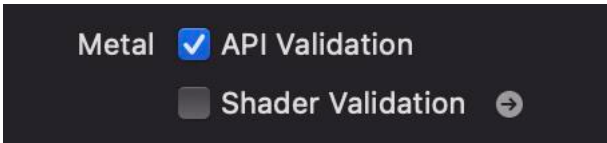
▲ Figure 3.73 Edit Scheme screen

Next, change GPU Frame Capture to "Metal" from the "Options" tab.



▲ Figure 3.74 Enable GPU Frame Capture

Finally, from the "Diagnostics" tab, enable "Api Validation" for Metal.



▲ Figure 3.75 Enable Api Validation

2. capture

Capture is performed by pressing the camera symbol from the debug bar during execution. Depending on the complexity of the scene, the first capture may take some time, so please be patient. Note that in Xcode13 or later, the icon has been changed to the Metal icon.

Xcode12 or earlier

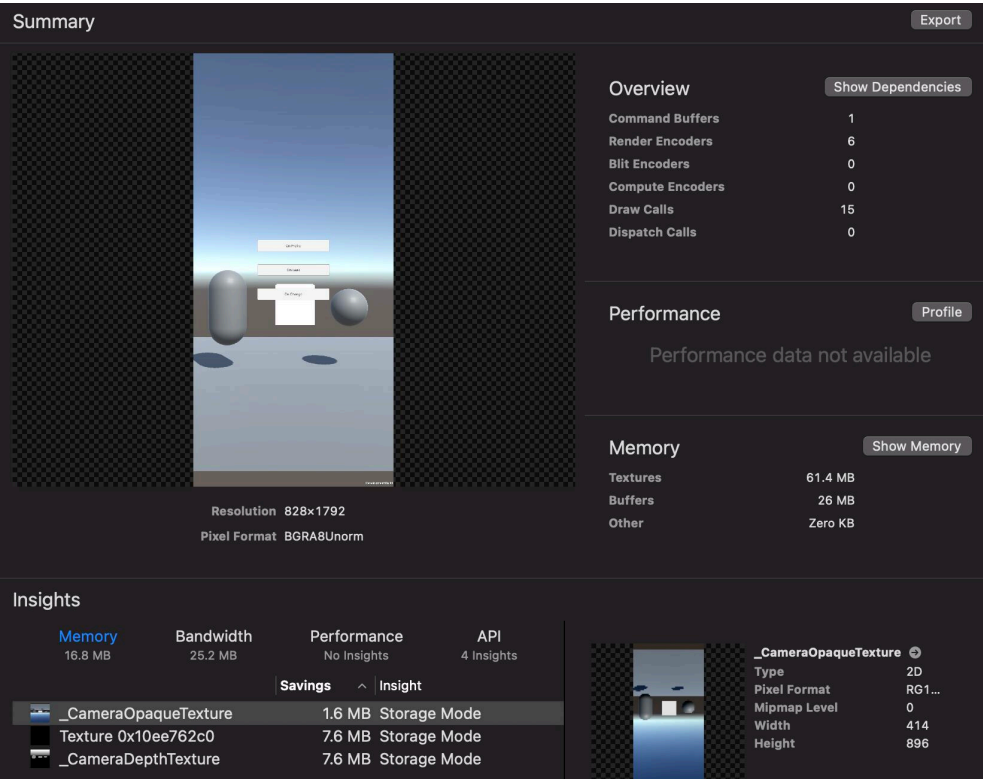


Xcode13 or later



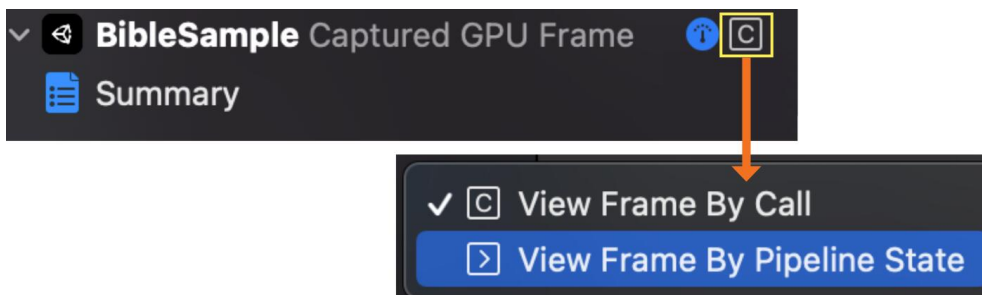
▲ Figure 3.76 GPU Frame Capture button

When the capture is completed, the following summary screen will be displayed.



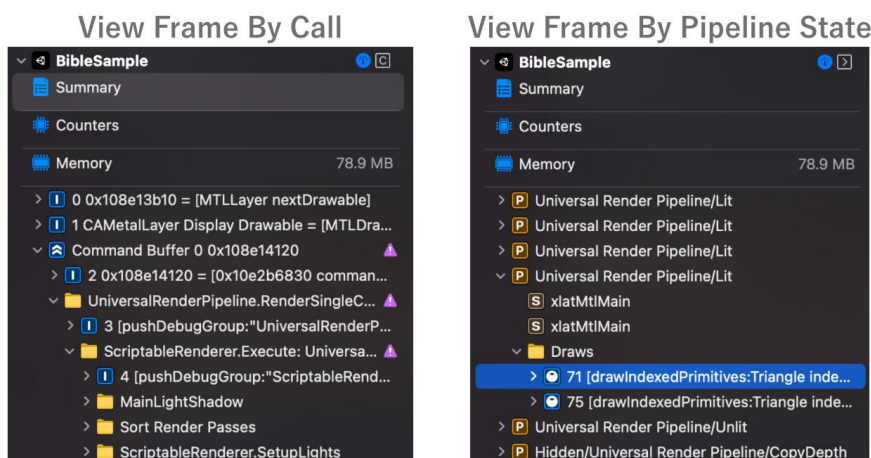
▲ Figure 3.77 Summary screen

From this summary screen, you can move to a screen where you can check details such as drawing dependencies and memory. The Navigator area displays commands related to drawing. There are "View Frame By Call" and "View Frame By Pipeline State".



▲ Figure 3.78 Changing the Display Method

In the By Call view, all drawing commands are listed in the order in which they were invoked. In the By Pipeline State view, all drawing commands are listed in the order in which they were invoked, which includes buffer settings and other preparations for drawing, so that a large number of commands are lined up. On the other hand, By Pipeline State lists only the drawing commands related to the geometry drawn by each shader. It is recommended to switch the display according to what you want to investigate.



▲ Figure 3.79 Differences in Display

By pressing any of the drawing commands in the Navigator area, you can check the properties used for that command. The properties include texture, buffer, sampler, shader functions, and geometry. Each property can be double-clicked to see

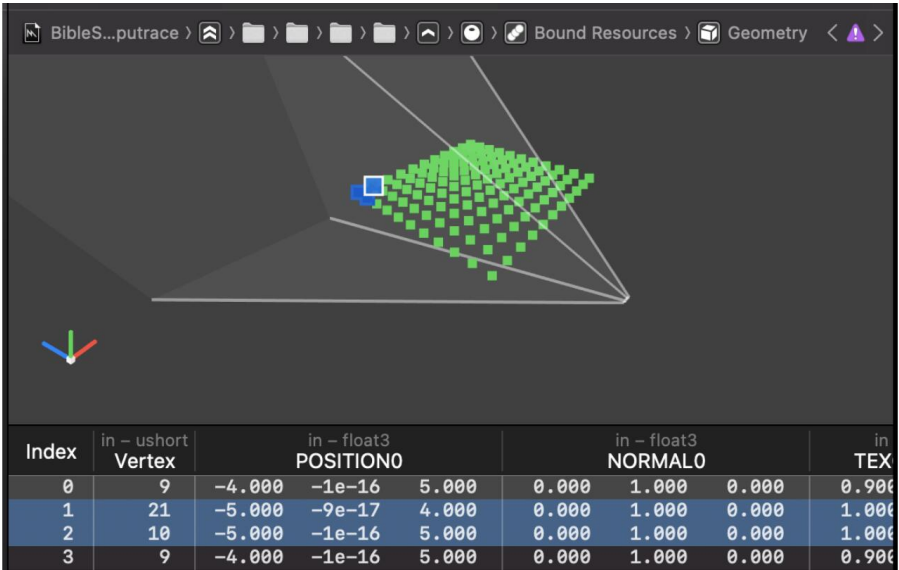
Chapter 3 Profiling Tools

the details. For example, you can see the shader code itself, whether the sampler is Repeat or Clamp, and so on.



▲ Figure 3.80 Drawing Command Details

Geometry properties not only display vertex information in a table format, but also allow you to move the camera to see the shape of the geometry.



▲ Figure 3.81 Geometry Viewer

Next, we will discuss "Profile" in the Performance column of the Summary screen. Clicking this button starts a more detailed analysis. When the analysis is finished, the time taken for drawing will be displayed in the Navigator area.

> P Hidden/Universal Render Pipeline/Blit	541.92 μ s
> P Universal Render Pipeline/Lit	437.30 μ s
> P Skybox/Procedural	398.49 μ s
> P Hidden/Universal Render Pipeline/CopyDepth	296.52 μ s

▲ Figure 3.82 Display after Profile

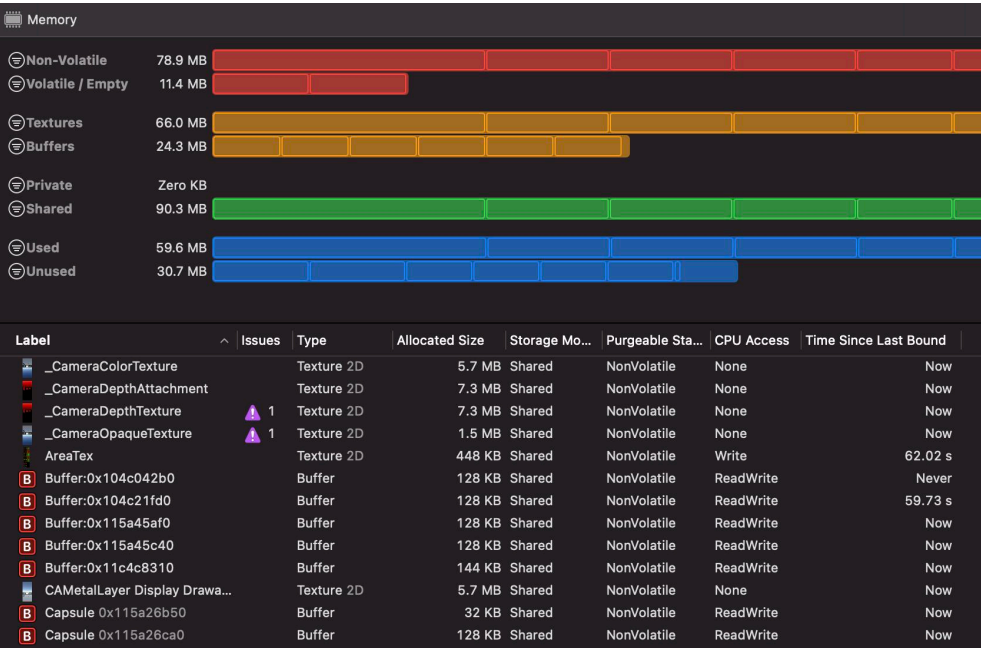
The results of the analysis can be viewed in more detail in the "Counters" screen. In this screen, you can graphically see the processing time for each drawing such as Vertex, Rasterized, Fragment, etc.



▲ Figure 3.83 Counters screen

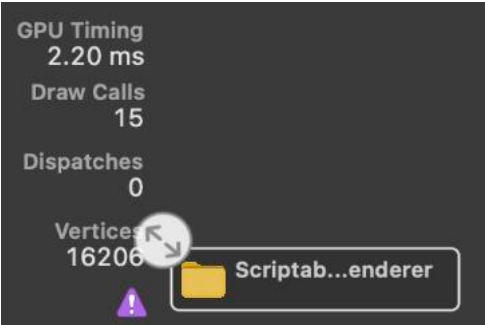
Next, "Show Memory" in the Memory column of the Summary screen is explained. Clicking this button will take you to a screen where you can check the resources used by the GPU. The information displayed is mainly textures and buffers. It is a good idea to check if there are any unnecessary items.

Chapter 3 Profiling Tools



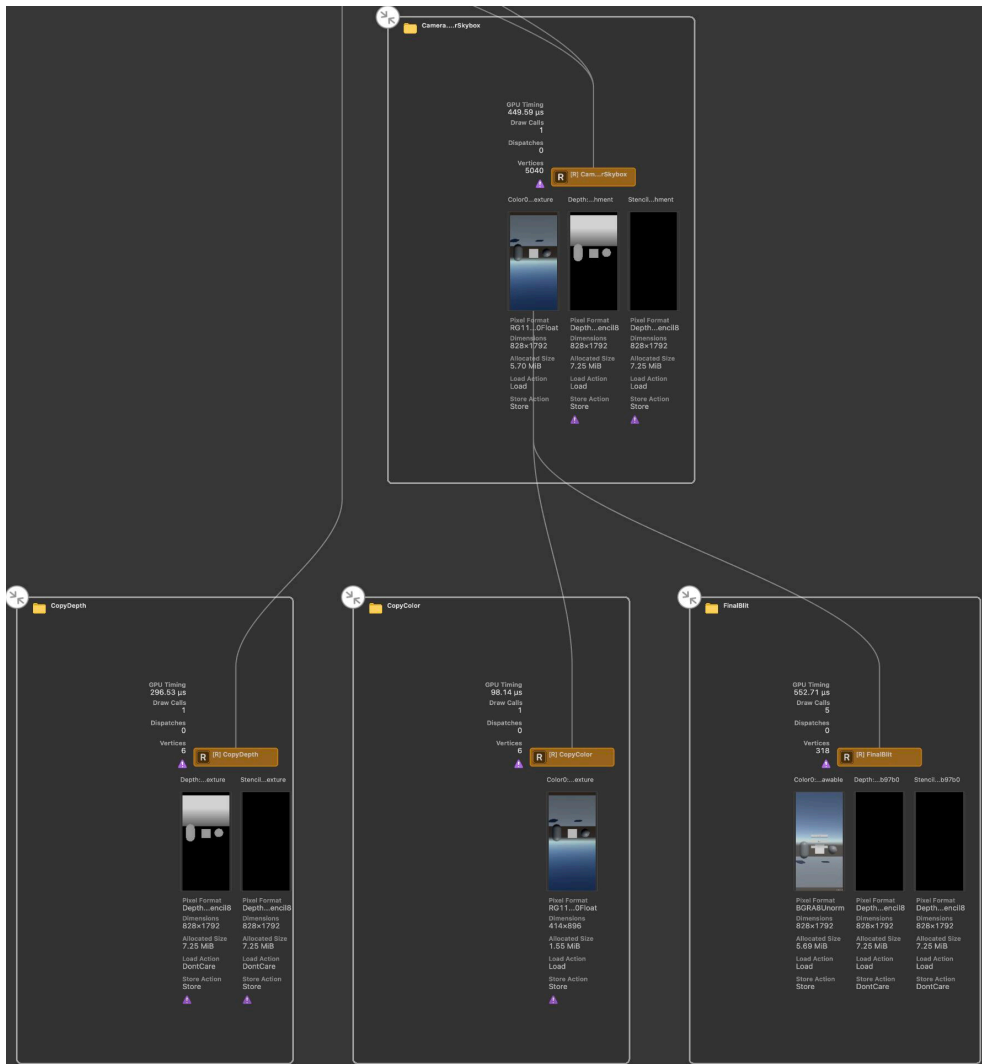
▲ Figure 3.84 GPU Resource Confirmation Screen

Finally, "Show Dependencies" in the Overview section of the Summary screen is explained. Clicking this button displays the dependencies for each render pass. When viewing the dependencies, click the button with the arrow pointing outward to open more dependencies below that level.



▲ Figure 3.85 Open Dependency button

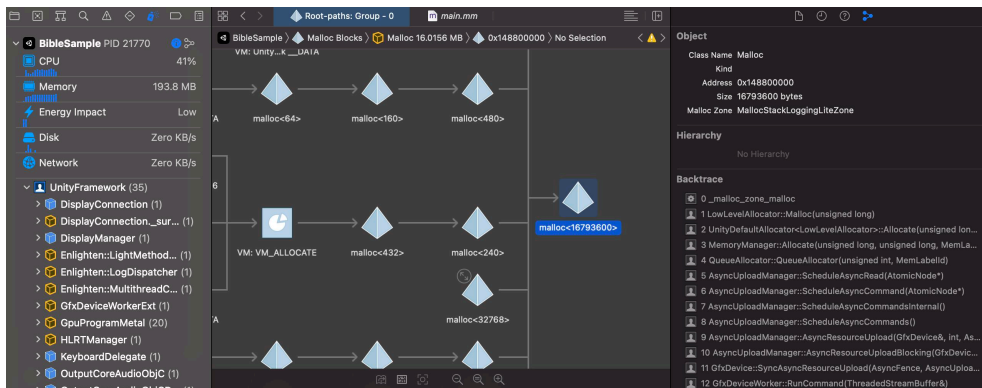
Use this screen when you want to see which drawings depend on what.



▲ Figure 3.86 With the hierarchy open

3.6.4 Memory Graph

This tool allows you to analyze the memory situation at the time of capture. The Navigator area on the left displays instances, and by selecting an instance, the reference relationships are displayed in a graph. The Inspector area on the right displays detailed information about the instance.

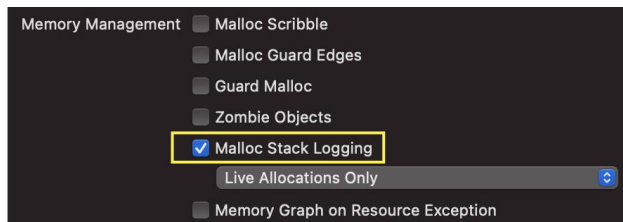


▲ Figure 3.87 MemoryGraph Capture Screen

This tool can be used to investigate memory usage of objects that cannot be measured in Unity, such as plug-ins. The following is an explanation of how to use this tool.

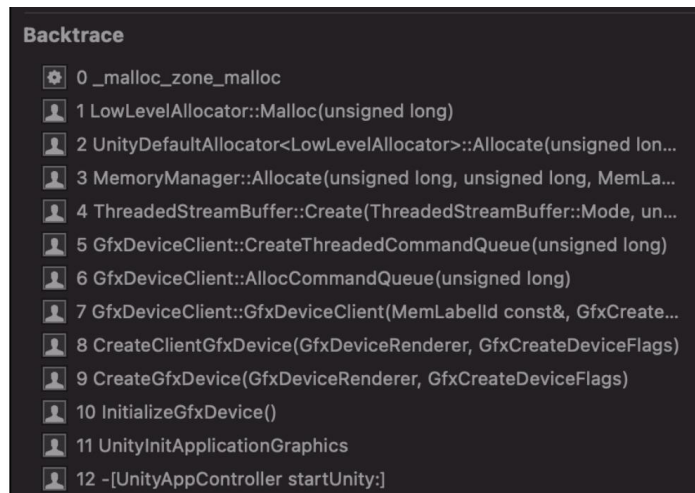
1. preliminary preparation

In order to obtain useful information from Memory Graph, it is necessary to edit the scheme. Open the scheme edit screen by clicking "Product -> Scheme -> Edit Scheme". Then, enable "Malloc Stack Logging" from the "Diagnostics" tab.



▲ Figure 3.88 Enable Malloc Stack Logging

By enabling this, Backtrace will be displayed in Inspector and you can see how it was allocated.



▲ Figure 3.89 Displaying Backtrace

2. capture

Capture is performed by pressing the branch-like icon from the debug bar while the application is running.




▲ Figure 3.90 Memory Graph Capture button

Chapter 3 Profiling Tools

Memory Graph can be saved as a file by clicking "File -> Export MemoryGraph". You can use the `vmmap` command, the `heap` command, and the `malloc_history` command to further investigate this file. If you are interested, please check it out. As an example, the summary display of the `vmmap` command is shown below, allowing you to grasp an overall picture that was difficult to grasp with the MemoryGraph command.

▼ List 3.5 vmmap summary command

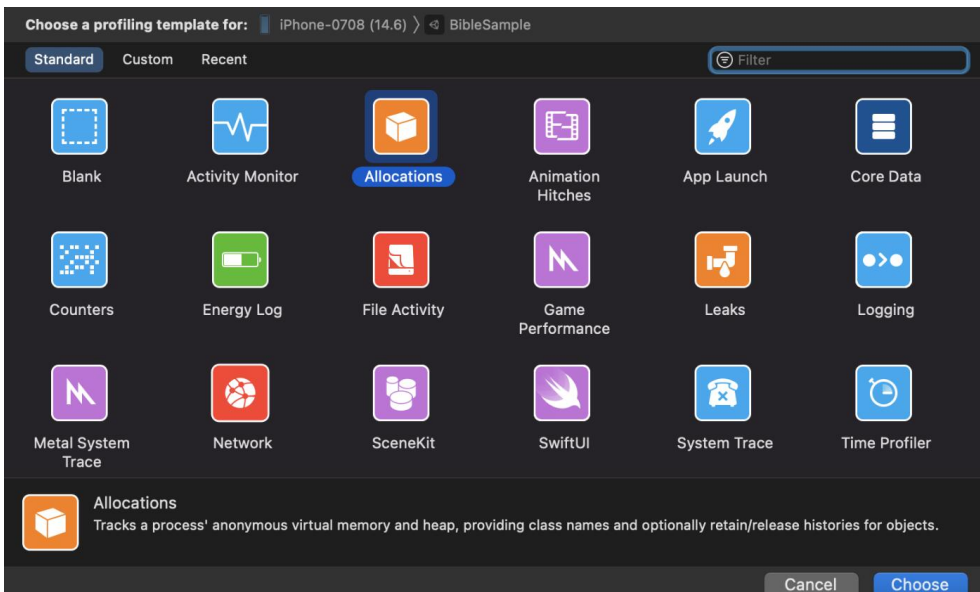
```
1: vmmap --summary hoge.memgraph
```

REGION TYPE	VIRTUAL SIZE	RESIDENT SIZE	DIRTY SIZE	SWAPPED SIZE	VOLATILE SIZE	NONVOL SIZE	EMPTY SIZE	REGION COUNT
=====	=====	=====	=====	=====	=====	=====	=====	=====
Activity Tracing	256K	32K	32K	0K	0K	32K	0K	1
CoreAnimation	32K	32K	32K	0K	0K	16K	0K	2
Foundation	16K	16K	16K	0K	0K	0K	0K	1
IOAccelerator	99.2M	78.4M	78.4M	0K	0K	78.4M	13.2M	98
IOKit	304K	304K	304K	0K	0K	0K	0K	19
IOSurface	17.1M	17.1M	17.1M	0K	0K	17.1M	0K	3
Image IO	16K	0K	0K	16K	0K	0K	0K	1
Kernel Alloc Once	32K	16K	16K	0K	0K	0K	0K	1
MALLOC guard page	192K	0K	0K	0K	0K	0K	0K	12
MALLOC metadata	336K	288K	288K	0K	0K	0K	0K	15
MALLOC_LARGE	69.8M	23.7M	23.7M	27.0M	0K	0K	0K	1215
MALLOC_LARGE metadata	256K	144K	144K	80K	0K	0K	0K	1
MALLOC_NANO	512.0M	304K	304K	32K	0K	0K	0K	1
MALLOC_SMALL	64.0M	5360K	3888K	9.9M	0K	0K	0K	8
MALLOC_SMALL (empty)	8192K	0K	0K	48K	0K	0K	0K	1
MALLOC_TINY	19.0M	8320K	8320K	2416K	0K	0K	0K	19
MALLOC_TINY (empty)	4096K	128K	128K	0K	0K	0K	0K	4
Performance tool data	4784K	4592K	4592K	192K	0K	0K	0K	43
								
=====	=====	=====	=====	=====	=====	=====	=====	=====
TOTAL	1.6G	355.9M	143.5M	41.6M	0K	95.6M	13.2M	4362

▲ Figure 3.91 MemoryGraph Summary display

3.7 Instruments

Xcode has a tool called Instruments that specializes in detailed measurement and analysis. To build Instruments, select "Product -> Analyze". Once completed, a screen will open to select a template for the measurement items as shown below.



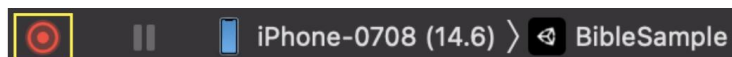
▲ Figure 3.92 Instruments template selection screen

As you can see from the large number of templates, Instruments can analyze a wide variety of content. In this section, we will focus on "Time Profiler" and "Allocations," which are frequently used.

3.7.1 Time Profiler

The Time Profiler is a tool for measuring code execution time. Like the CPU module in the Unity Profiler, it is used to improve processing time.

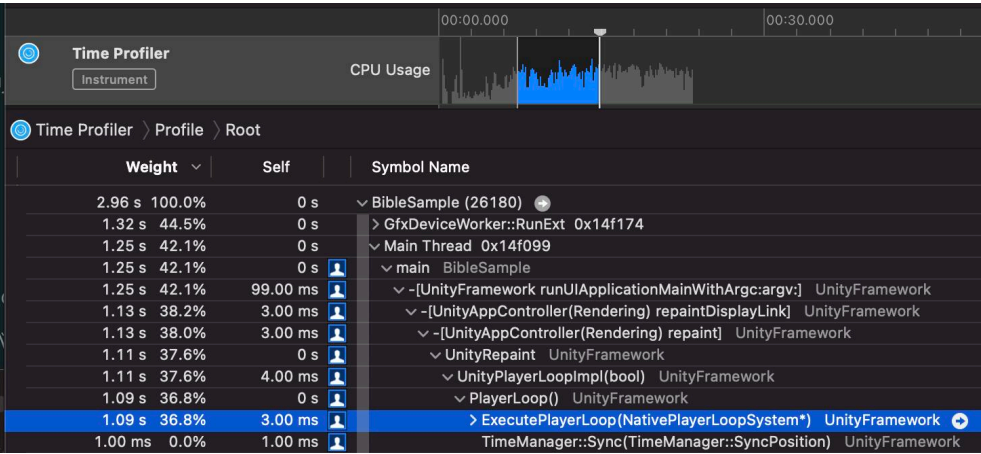
To start the measurement, you need to click on the record button marked with a red circle in the Time Profiler toolbar.



▲ Figure 3.93 Start Record button

When the measurement is performed, the display will look like Figure 3.94.

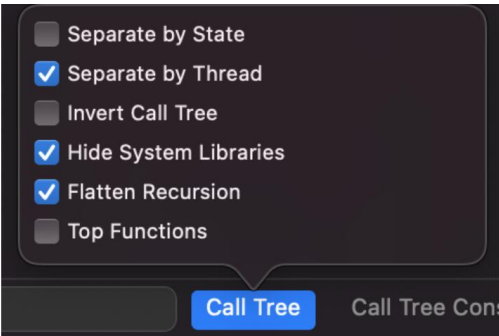
Chapter 3 Profiling Tools



▲ Figure 3.94 Measurement Result

Unlike the Unity Profiler, we will be analyzing not in frames, but in segments. The Tree View at the bottom shows the processing time within the interval. When optimizing the processing time of game logic, it is recommended to analyze the processing below the PlayerLoop in the Tree View.

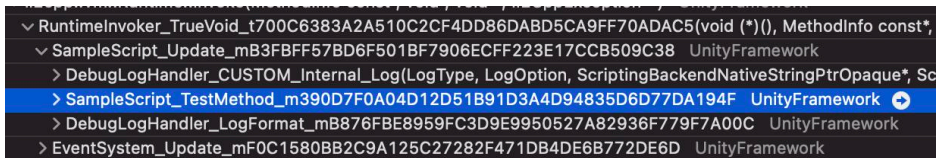
To make the Tree View display easier to read, you should set the Call Trees setting at the bottom of Xcode like Figure 3.95. In particular, checking the Hide System Libraries checkbox hides inaccessible system code, making it easier to investigate.



▲ Figure 3.95 Call Trees Settings

In this way, processing times can be analyzed and optimized.

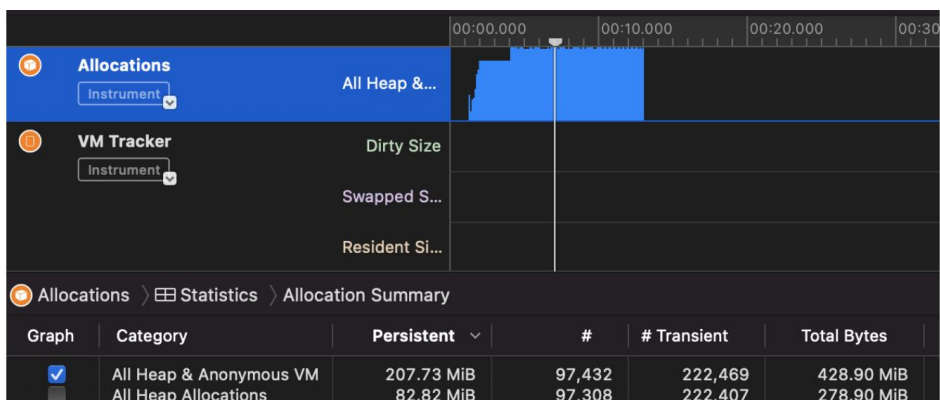
The symbol names in the Time Profiler differ from those in the Unity Profiler. The symbol names in the Time Profiler are different from those in the Unity Profiler, but they are still the same: "class_name_function_name_random_string".



▲ Figure 3.96 Symbol name in Time Profiler

3.7.2 Allocations

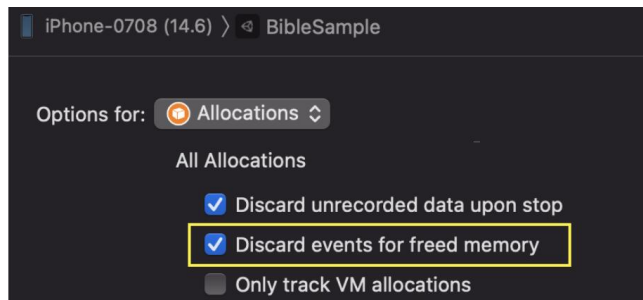
Allocations is a tool for measuring memory usage. It is used to improve memory leakage and usage.



▲ Figure 3.97 Allocations measurement screen

Before measuring, open "File -> Recording Options" and check "Discard events for freed memory".

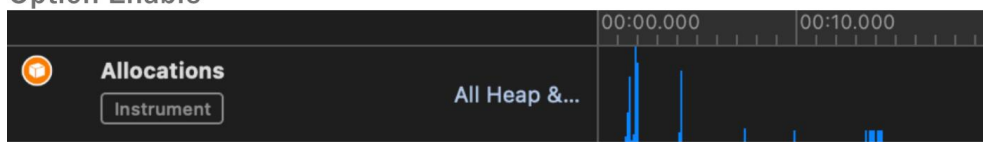
Chapter 3 Profiling Tools



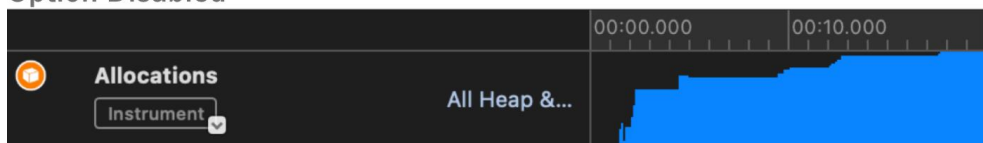
▲ Figure 3.98 Option setting screen

If this option is enabled, the recording will be discarded when memory is freed.

Option Enable



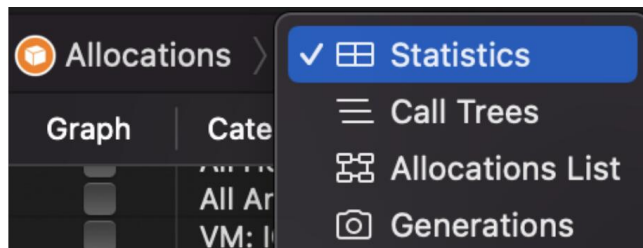
Option Disabled



▲ Figure 3.99 Difference by Option Setting

Figure 3.99 As you can see in Figure 1, the appearance changes significantly with and without options. With the option, lines are recorded only when memory is allocated. Also, the recorded lines are discarded when the allocated area is released. In other words, by setting this option, if a line remains in memory, it has not been released from memory. For example, in a design where memory is released by scene transitions, if many lines remain in the scene section before the transition, there is a suspicion of a memory leak. In such a case, use the Tree View to follow the details.

The Tree View at the bottom of the screen displays the details of the specified range, similar to the Time Profiler. The Tree View can be displayed in four different ways.



▲ Figure 3.100 Selecting a display method

The most recommended display method is Call Trees. This allows you to follow which code caused the allocation. There are Call Trees display options at the bottom of the screen, and you can set options such as Hide System Libraries in the same way as Figure 3.95 introduced in the Time Profiler. Figure 3.101 Now we have captured the Call Trees display. You can see that 12.05MB of allocation is generated by SampleScript's OnClicked.

12.06 MB	49.0%	49	▼ Main Thread 0x2bcb8c0
12.06 MB	49.0%	49	▼ main BibleSample
12.06 MB	49.0%	49	▼ -[UnityFramework runUIApplicationMainWithArgc:argv:] UnityFramework
12.05 MB	49.0%	39	▼ -[UnityAppController(Rendering) repaintDisplayLink] UnityFramework
12.05 MB	49.0%	39	▼ -[UnityAppController(Rendering) repaint] UnityFramework
12.05 MB	49.0%	39	▼ UnityRepaint UnityFramework
12.05 MB	49.0%	39	▼ UnityPlayerLoopImpl(bool) UnityFramework
12.05 MB	49.0%	39	▼ PlayerLoop() UnityFramework

12.05 MB	49.0%	36	▼ UnityAction_Invoke_mFFF1FFE59D8285F8A8135
12.05 MB	49.0%	20	> SampleScript_OnClickedLeak_m9D4818A571B
2.98 KB	0.0%	16	> MonoBehaviour_CUSTOM_StartCoroutineManag

▲ Figure 3.101 Call Tree display

Finally, let me introduce a feature called Generations. At the bottom of Xcode, there is a button called "Mark Generations."



▲ Figure 3.102 Mark Generation button

When this button is pressed, the memory at that timing is stored. After that, press-

Chapter 3 Profiling Tools

ing the "Mark Generations" button again will record the amount of memory newly allocated compared to the previous data.

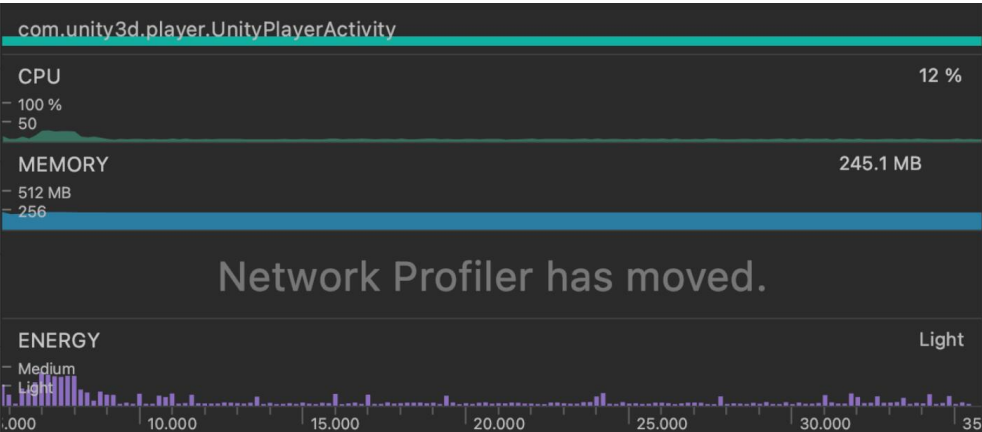
Allocations > Generations > All Generations			
Snapshot	Timestamp	Growth	# Persistent
> Generation A	00:15.602.827	168.47 MiB	96,330
> Generation B	00:20.858.176	75.71 MiB	6,219
> Generation C	00:24.390.265	256 Bytes	1

▲ Figure 3.103 Generations

Figure 3.103 Each Generation in is displayed in a Call Tree format so that you can follow what caused the memory increase.

3.8 Android Studio

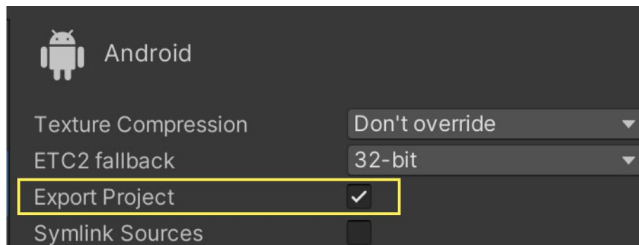
Android Studio is an integrated development environment tool for Android. This tool allows you to measure the status of your application. There are four profileable items: CPU, Memory, Network, and Energy. In this section, we will first introduce the profiling method and then explain the measurement items of CPU and Memory.



▲ Figure 3.104 Profile screen

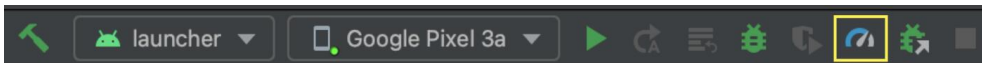
3.8.1 Profile Method

There are two ways to profile. The first is to build and profile via Android Studio. In this method, first export the Android Studio project from Unity. In the Build Settings, check the "Export Project" checkbox and build the project.



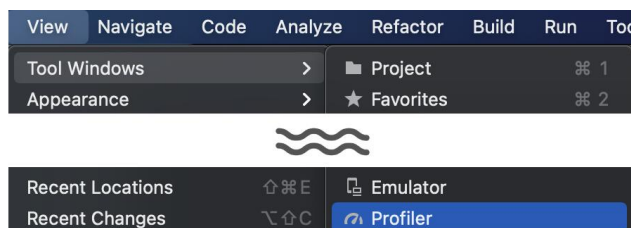
▲ Figure 3.105 Export Project

Next, open the exported project in Android Studio. Then, with the Android device connected, press the gauge-like icon in the upper right corner to start the build. After the build is complete, the application will launch and the profile will start.



▲ Figure 3.106 Profile start icon

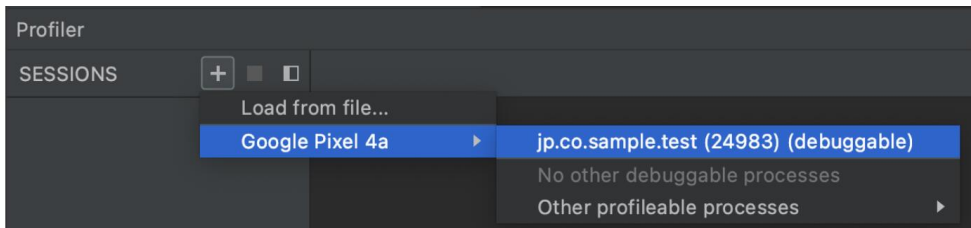
The second method is to attach the running process to the debugger and measure it. First, open the Android Profiler from "View -> Tool Windows -> Profiler" in the Android Studio menu.



▲ Figure 3.107 Open Android Profiler

Chapter 3 Profiling Tools

Next, open the Profiler and click on **SESSIONS** in the Profiler. To connect a session, the application to be measured must be running. Also, the **binary must be a Development Build**. Once the session is connected, the profile will start.



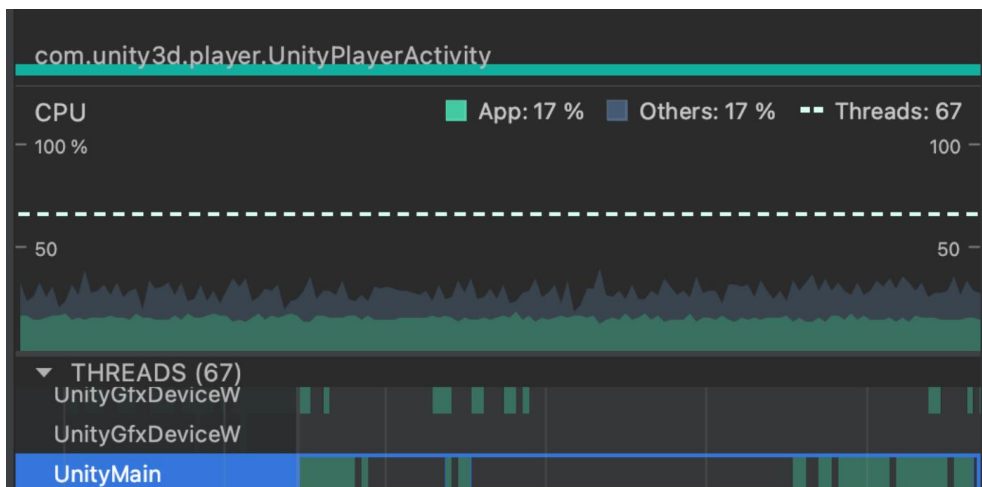
▲ Figure 3.108 Select the SESSION to profile

The second method of attaching to the debugger is good to keep in mind because it does not require exporting the project and can be used easily.

Strictly speaking, you need to configure debuggable and profileable settings in AndroidManifest.xml, not Development Build in Unity. In Unity, debuggable is automatically set to true when you do a Development Build.

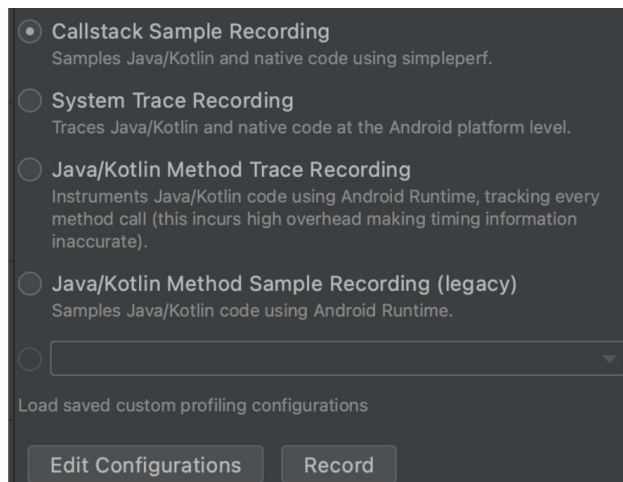
3.8.2 CPU Measurement

The CPU measurement screen looks like Figure 3.109. This screen alone does not tell you what is consuming how much processing time. To see more details, you need to select the threads you want to see in detail.



▲ Figure 3.109 CPU measurement top screen, thread selection

After selecting a thread, press the Record button to measure the thread's call stack. Figure 3.110 There are several measurement types like "Callstack Sample Recording", but "Callstack Sample Recording" will be fine.

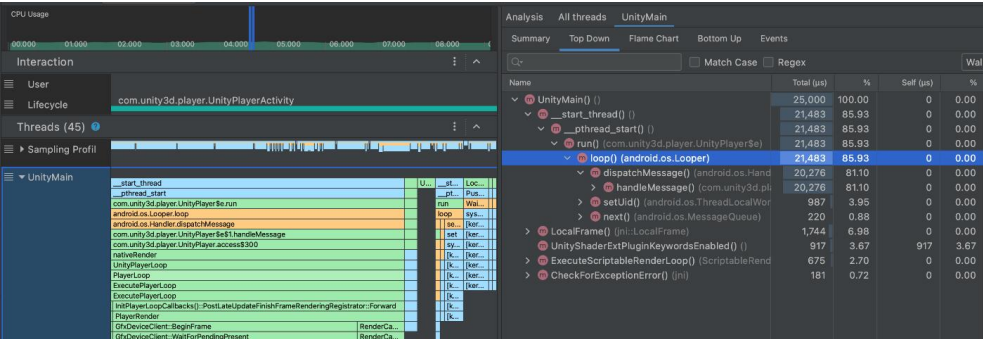


▲ Figure 3.110 Record start button

Clicking the Stop button will end the measurement and display the results. The

Chapter 3 Profiling Tools

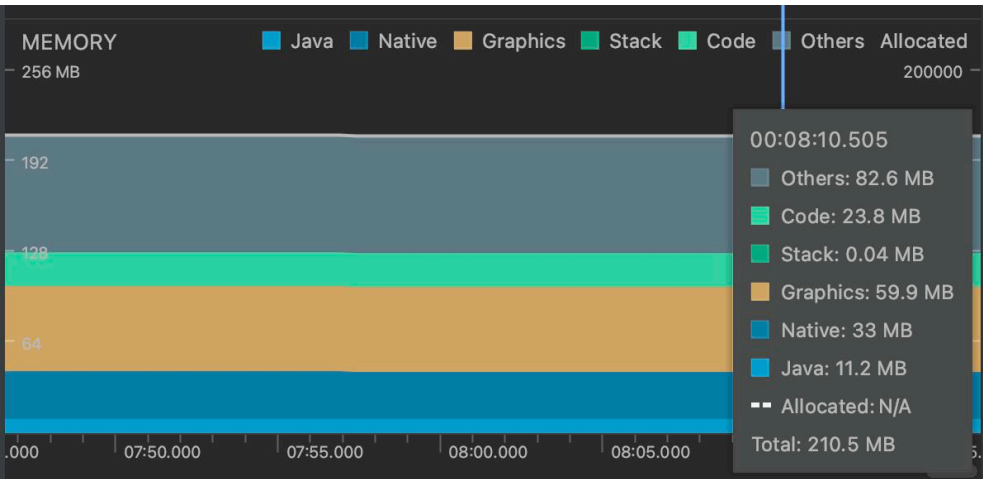
result screen will look like the CPU module of the Unity Profiler.



▲ Figure 3.111 Call Stack Measurement Result Screen

3.8.3 Memory Measurement

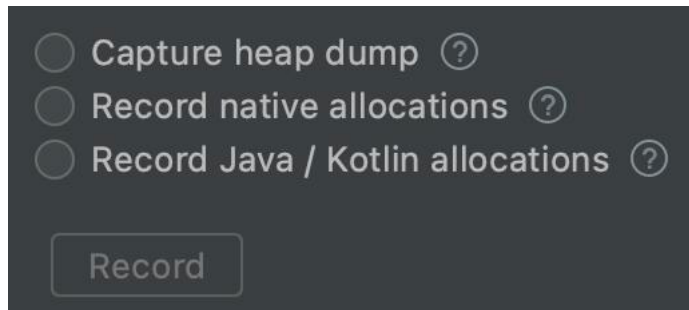
The Memory measurement screen looks like Figure 3.112. The memory breakdown cannot be seen on this screen.



▲ Figure 3.112 Memory measurement screen

If you want to see the breakdown of memory, you need to perform an additional measurement. There are three measurement methods. Capture heap dump" can

acquire the memory information at the timing when it is pressed. Other buttons are for analyzing allocations during the measurement section.



▲ Figure 3.113 Memory measurement options

As an example, we have captured the measurement results of Heap Dump at Figure 3.114. The granularity is a bit coarse for detailed analysis, so it may be challenging.

617	0	48,632	53,605	3,256,122	3,787,319
Classes	Leaks	Count	Native Size	Shallow Size	Retained Size
Class Name		Allocations	Native Size	Shallow Size	Retained Size
app heap		48,632	53,605	3,256,122	3,787,319
byte[]		7,465	0	1,990,508	1,990,508
Object[] (java.lang)		7,196	0	184,476	258,747
Method (java.lang.reflect)		5,408	0	216,320	216,320
int[]		2,513	0	179,120	179,120

Instance List - byte[]				
Instance	Depth	Native ...	Shallow ...	Retain...
byte[]@377946320 (0x168700d0)	-	0	8,744	8,744
byte[]@320700248 (0x131d7f58)	-	0	8,192	8,192
byte[]@319033008 (0x13040eb0)	-	0	8,192	8,192
byte[]@318242816 (0x12f80000)	-	0	8,192	8,192
byte[]@320962384 (0x13215840)	-	0	8,192	8,192
byte[]@317890184 (0x12f29e88)	-	0	8,192	8,192

Instance Details - byte[]@377946320 (0x168700d0)				
References				
Show nearest GC root only				
Reference	Depth	Native...	Shallow...	Retained...
byte[]@377946320 (0x168700d0)	-	0	8,744	8,744
keyBytes in ICUResourceBun	-	0	55	55

▲ Figure 3.114 Heap Dump Results

3.9 RenderDoc

RenderDoc is an open source, free, high-quality graphics debugger tool. The tool is currently available for Windows and Linux, but not for Mac. Graphics APIs supported include Vulkan, OpenGL(ES), D3D11, and D3D12. Therefore, it can be used on

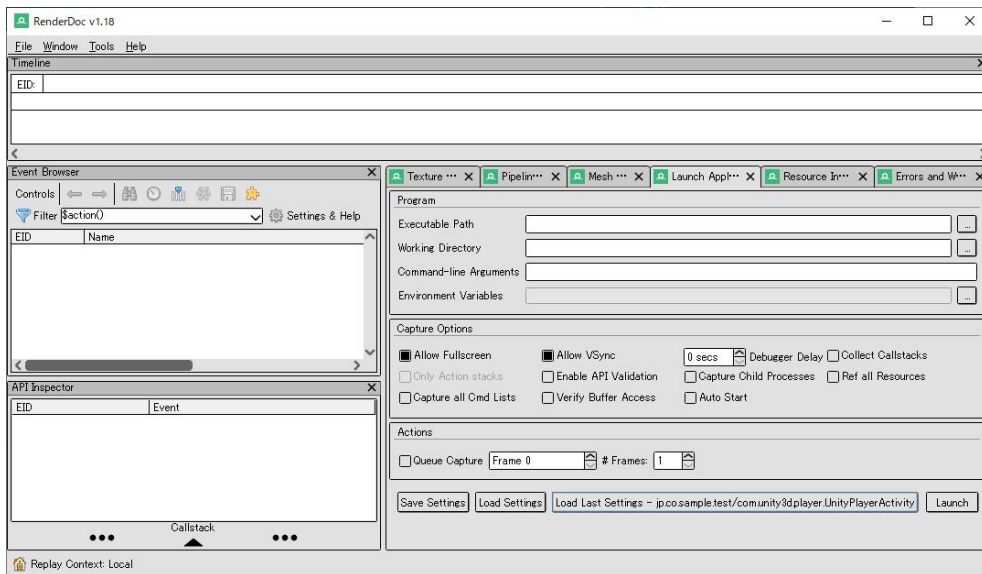
Chapter 3 Profiling Tools

Android, but not on iOS.

In this section, we will actually profile an Android application. Note, however, that there are some limitations to Android profiling. First, Android OS version 6.0 or later is required. And the application to be profiled must have Debuggable enabled. This is no problem if Development Build is selected at build time. The version of RenderDoc used for the profile is v1.18.

3.9.1 Measurement Method

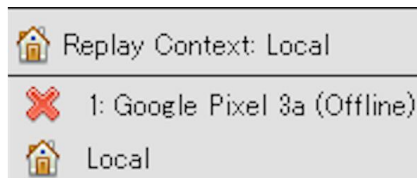
First, prepare RenderDoc. Download the installer from the official website ^{*3} and install the tool. After installation, open the RenderDoc tool.



▲ Figure 3.115 Screen after launching RenderDoc

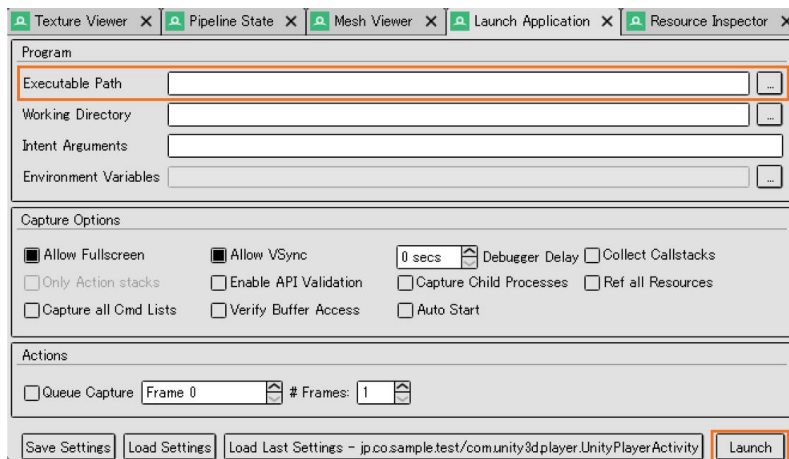
Next, connect your Android device to RenderDoc. Click the house symbol in the lower left corner to display the list of devices connected to the PC. Select the device you want to measure from the list.

^{*3} <https://renderdoc.org/>



▲ Figure 3.116 Connecting to a device

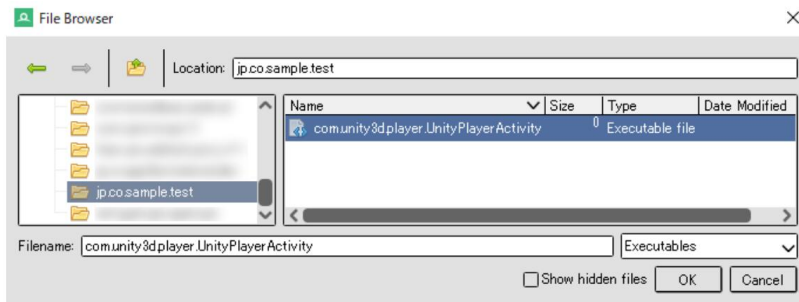
Next, select the application to be launched from the connected device. Select Launch Application from the tabs on the right side and choose the application to run from the Executable Path.



▲ Figure 3.117 Launch Application tab Select the application to run

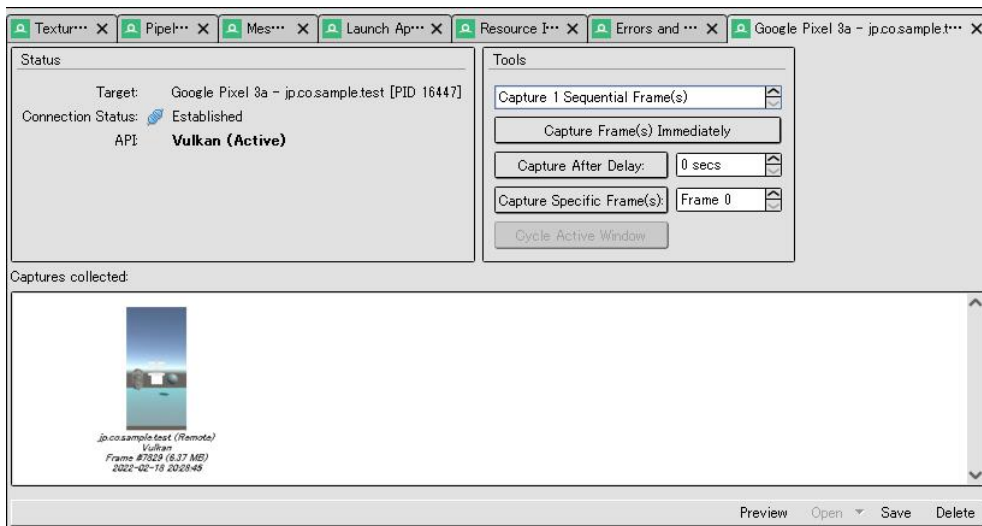
A File Browser window will open. Find the Package Name for this measurement and select Activity.

Chapter 3 Profiling Tools



▲ Figure 3.118 Select the application to be measured

Finally, from the Launch Application tab, click the Launch button to launch the application on the device. In addition, a new tab for measurement will be added on the RenderDoc.

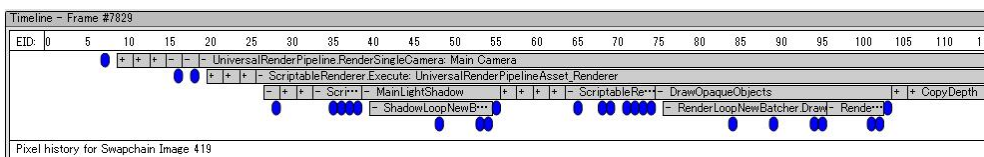


▲ Figure 3.119 Tab for measurement

"Capture Frame(s) Immediately" will capture frame data, which will be listed in the "Capture collected" tab. Double-click on this data to open the captured data.

3.9.2 How to View Capture Data

RenderDoc has a variety of functions, but in this section we will focus on the most important ones. First, a timeline of captured frames is displayed at the top of the screen. This allows you to visually capture the order in which each drawing command was performed.



▲ Figure 3.120 Timeline

Next is the Event Browser. Each command is listed here in order from the top.

EID	Name	Duration (μs)
64-73	ScriptableRenderPass.Configure	0.00
75-102	DrawOpaqueObjects	4.53125
76-94	RenderLoopNewBatch.Draw	3.75
84	vkCmdDrawIndexed(600, 1)	2.13542
89	vkCmdDrawIndexed(2304, 1)	0.83333
94	vkCmdDrawIndexed(36, 1)	0.78125
96-101	RenderLoop.Draw	0.78125
101	vkCmdDrawIndexed(2496, 1)	0.78125
104	ScriptableRenderPass.Configure	0.00

▲ Figure 3.121 Event Browser

Clicking the "clock symbol" at the top of the Event Browser displays the processing time for each command in the "Duration" column. The processing time varies depending on the timing of the measurement, so it is best to consider it as a rough estimate. The breakdown of the DrawOpaqueObjects command shows that three commands are batch processed and only one is drawn out of batch.

Next, let's look at the tabs on the right side of the window. In this tab, there is a window where you can check detailed information about the command selected in

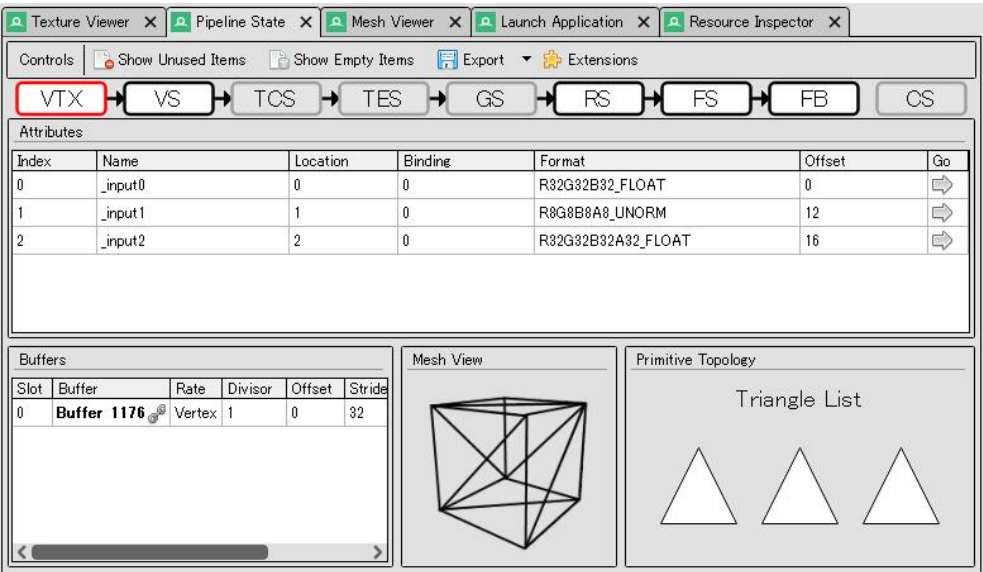
Chapter 3 Profiling Tools

the Event Browser. The three most important windows are the Mesh Viewer, Texture Viewer, and Pipeline State.



▲ Figure 3.122 Each window

First, let's look at Pipeline State. Pipeline State allows you to see what parameters were used in each shader stage before the object was rendered to the screen. You can also view the shaders used and their contents.



▲ Figure 3.123 Pipeline State

The stage names displayed in the Pipeline State are abbreviated, so the official names are summarized at Table 3.7.

▼ Table 3.7 Official name of PipelineState

Stage Name	Official name
VTX	Vertex Input
VS	Vertex Shader
TCS	Tessellation Control Shader
TES	Tessellation Evaluation Shader
GS	Geometry Shader
Rasterizer	Rasterizer
FS	Fragment Shader
Frame Buffer	Frame Buffer
Frame Buffer	Compute Shader

Figure 3.123 The VTX stage is selected at , where you can see the topology and vertex input data. Other FB stages at Figure 3.124 allow you to see details such as the state of the output destination texture and Blend State.

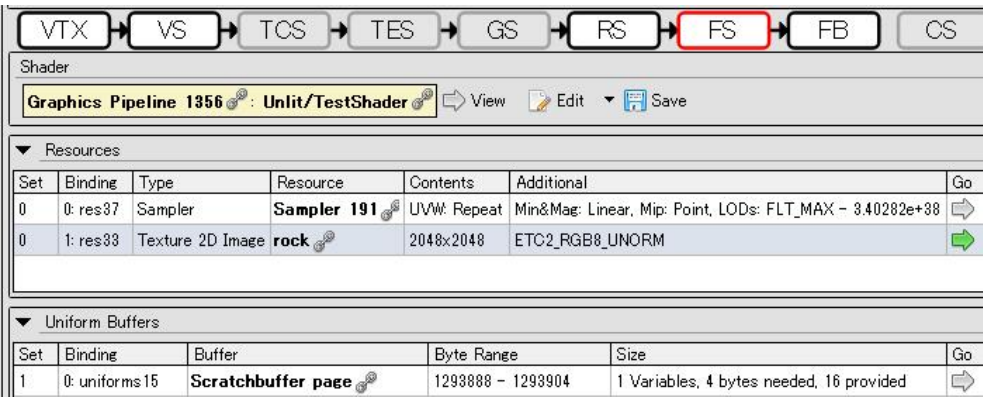
Target Blends								
Slot	Enabled	Col Src	Col Dst	Col Op	Alpha Src	Alpha Dst	Alpha Op	Write Mask
0	False	One	Zero	Add	One	Zero	Add	RGBA

Blend State		Depth State		Stencil State					
Blend Factor:	1.00, 1.00, 1.00, 1.00	Enabled:	✓	Face	Func	Fail Op	Depth Fail Op	Pass Op	Write Mask
Logic Op:	—	Func:	Greater Equal	Front	—	—	—	—	—
		Write:	✓	Back	—	—	—	—	—
		Bounds:	✗						

▲ Figure 3.124 FB (Frame Buffer) State

You can also check the FS stage at Figure 3.125 to see the textures and parameters used in the fragment shader.

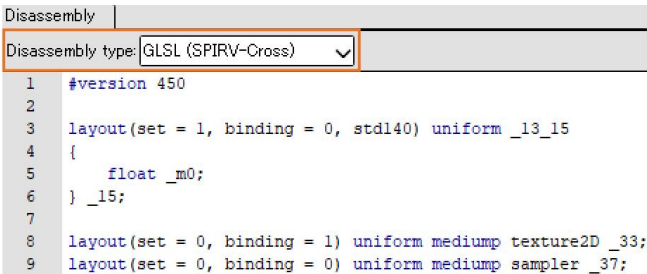
Chapter 3 Profiling Tools



▲ Figure 3.125 State of FS (Fragment Shader)

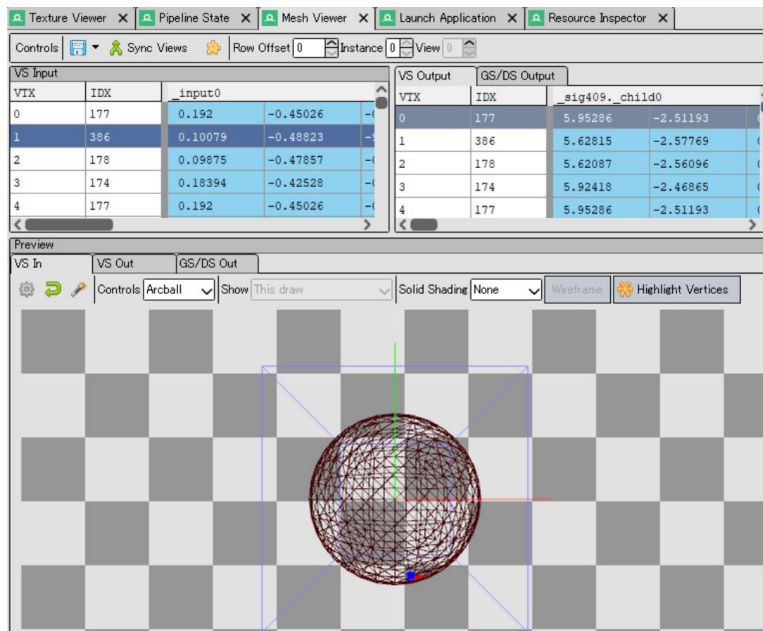
Resources in the center of the FS stage shows the textures and samplers used. Uniform Buffers at the bottom of the FS stage shows the CBuffer. This CBuffer contains numerical properties such as float and color. To the right of each item, there is a "Go" arrow icon, which can be pressed to see the details of the data.

The shader used is shown in the upper part of the FS stage, and the shader code can be viewed by pressing View. Disassembly type GLSL is recommended to make the display easier to understand.



▲ Figure 3.126 Confirmation of Shader Code

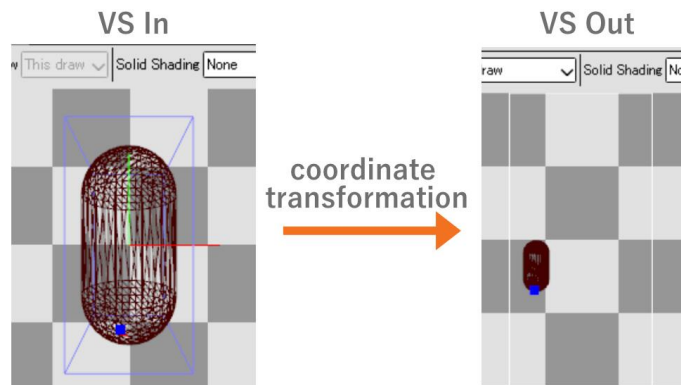
Next is the Mesh Viewer. This function allows you to visually view mesh information, which is useful for optimization and debugging.



▲ Figure 3.127 Mesh Viewer

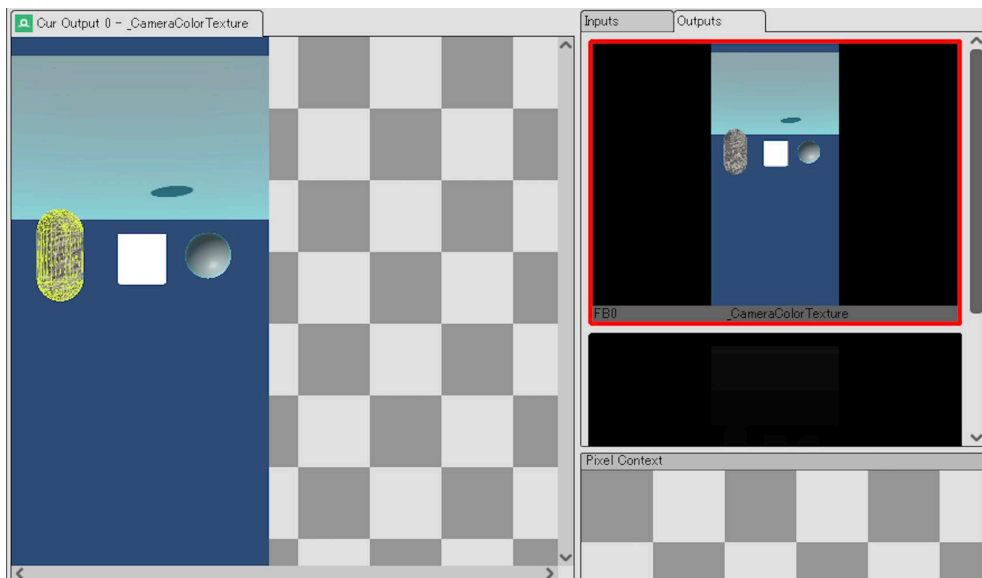
The upper part of the Mesh Viewer shows mesh vertex information in a table format. The lower part of the Mesh Viewer has a preview screen where you can move the camera to check the shape of the mesh. Both tabs are divided into In and Out tabs, so you can see how the values and appearance have changed before and after the conversion.

Chapter 3 Profiling Tools



▲ Figure 3.128 Preview display of In and Out in Mesh Viewer

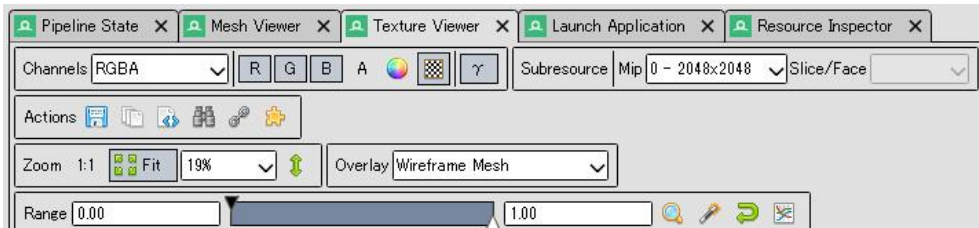
Finally, there is the Texture Viewer. This screen shows the "texture used for input" and "output result" of the command selected in the Event Browser.



▲ Figure 3.129 Texture Viewer Texture Confirmation Screen

In the area on the right side of the screen, you can check the input and output textures. Clicking on the displayed texture will reflect it in the area on the left side of the screen. The left side of the screen not only displays the texture, but also allows

you to filter the color channels and apply toolbar settings.



▲ Figure 3.130 Texture Viewer Toolbar

Figure 3.129 In the example above, "Wireframe Mesh" was selected for the Overlay, so the object drawn with this command has a yellow wireframe display, making it easy to see visually.

Texture Viewer also has a feature called Pixel Context. This function allows the user to view the drawing history of selected pixels. The history allows the user to determine how often a pixel has been filled. This is a useful feature for overdraw investigation and optimization. However, since it is on a per-pixel basis, it is not suitable for investigating overdraw on a global basis. To investigate, right-click on the area you want to investigate on the left side of Figure 3.129, and the location will be reflected in the Pixel Context.



▲ Figure 3.131 Reflection in Pixel Context

Next, click the History button in the Pixel Context to see the drawing history of the pixel.

Chapter 3 Profiling Tools

Pixel History on _CameraColorTexture for (195, 734)			
Preview colours displayed in visible range 0.00 - 1.00 with RGB channels visible.			
Double click to jump to an event. Right click to debug an event, or hide failed events.			
Event	Tex Before	Tex After	
<ul style="list-style-type: none"> > UniversalRenderPipeline.RenderSingleCamera: Main Camera > ScriptableRenderer.Execute: UniversalRenderPipelineAsset_Renderer > ScriptableRenderPass.Configure 			
EID 73 vkCmdBeginRenderPass(C=Clear, DS=Clear) 1 Fragments touching pixel	R: 0.00 G: 0.00 B: 0.00 D: 0.00 S: 0x00	R: 0.19141 G: 0.30078 B: 0.46875 D: 0.00 S: 0x00	
<ul style="list-style-type: none"> > UniversalRenderPipeline.RenderSingleCamera: Main Camera ... > DrawOpaqueObjects > RenderLoopNewBatcher.Draw 	Tex Before	Tex After	
EID 84 vkCmdDrawIndexed(600, 1) Depth test failed 1 Fragments touching pixel	R: 0.19141 G: 0.30078 B: 0.46875 D: 0.00 S: 0x00	R: 0.54688 G: 0.82031 B: 0.84375 D: 0.02157 S: 0x00	
<ul style="list-style-type: none"> > UniversalRenderPipeline.RenderSingleCamera: Main Camera ... > DrawOpaqueObjects > RenderLoop.Draw 	Tex Before	Tex After	
EID 101 vkCmdDrawIndexed(2496, 1) 1 Fragments touching pixel	R: 0.54688 G: 0.82031 B: 0.84375 D: 0.02157 S: 0x00	R: 0.51563 G: 0.51563 B: 0.51563 D: 0.03177 S: 0x00	
<ul style="list-style-type: none"> > UniversalRenderPipeline.RenderSingleCamera: Main Camera > ScriptableRenderer.Execute: UniversalRenderPipelineAsset_Renderer > Camera.RenderSkybox 	Tex Before	Tex After	
EID 137 vkCmdDraw(5040, 1) Depth test failed 1 Fragments touching pixel	R: 0.51563 G: 0.51563 B: 0.51563 D: 0.03177 S: 0x00	R: 0.51563 G: 0.51563 B: 0.51563 D: 0.03177 S: 0x00	

▲ Figure 3.132 Pixel Drawing History

Figure 3.132 In the following section, there are four histories. The green line indicates that the pixel passed all the pipeline tests, such as the depth test, and was painted. If some of the tests failed and the pixel was not rendered, it will be red. In the captured image, the screen clearing process and capsule drawing were successful, while the Plane and Skybox failed the depth test.



PERFORMANCE TUNING BIBLE

CHAPTER 04

第4章

**Tuning Practice
— Asset —**

CyberAgent Smartphone Games & Entertainment

Chapter 4

Tuning Practice - Asset

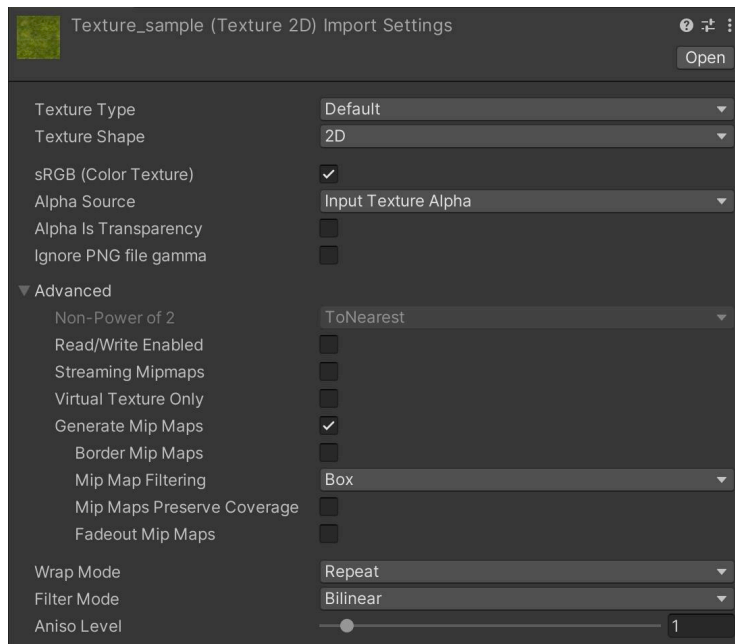
Game production involves handling a large number of different types of assets such as textures, meshes, animations, and sounds. This chapter provides practical knowledge about these assets, including settings to keep in mind when tuning performance.

4.1 Texture

Image data, which is the source of textures, is an indispensable part of game production. On the other hand, it consumes a relatively large amount of memory, so it must be configured appropriately.

4.1.1 Import Settings

Figure 4.1 is the import settings for textures in Unity.



▲ Figure 4.1 Texture Settings

Here are some of the most important ones to keep in mind.

4.1.2 Read/Write

This option is disabled by default. If disabled, textures are only expanded in GPU memory. If enabled, it will be copied not only to GPU memory but also to main memory, thus doubling the consumption. Therefore, if you do not use APIs such as `Texture.GetPixel` or `Texture.SetPixel` and only use Shader to access textures, be sure to disable them.

Also, for textures generated at runtime, set `makeNoLongerReadable` to true as shown at List 4.1 to avoid copying to main memory.

▼ List 4.1 Setting makeNoLongerReadable

```
1: texture2D.Apply(updateMipmaps, makeNoLongerReadable: true)
```

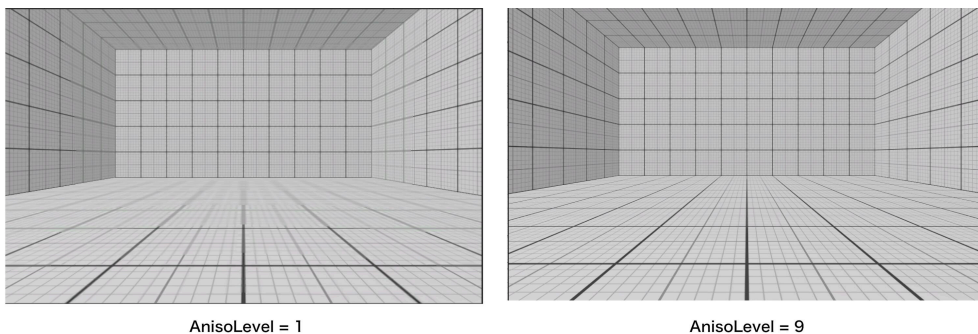
Since transferring textures from GPU memory to main memory is time-consuming, performance is improved by deploying textures to both if they are readable.

4.1.3 Generate Mip Maps

Enabling the Mip Map setting increases texture memory usage by about 1.3 times. This setting is generally used for 3D objects to reduce jaggies and texture transfer for distant objects. It is basically unnecessary for 2D sprites and UI images, so it should be disabled.

4.1.4 Aniso Level

Aniso Level is a function to render textures without blurring when objects are rendered at shallow angles. This function is mainly used for objects that extend far, such as the ground or floor. The higher the Aniso Level value, the more benefit it provides, but at a higher processing cost.

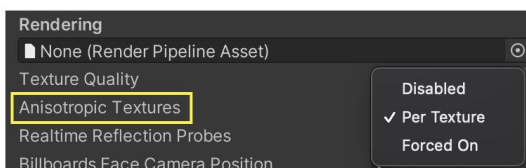


▲ Figure 4.2 Aniso Level adaptation image

The Aniso Level can be set from 0 to 16, but it has a slightly special specification.

- 0: Always disabled regardless of project settings
- 1: Basically disabled. However, if the project setting is Forced On, the value is clamped to 9~16.
- Otherwise: Set at that value

When textures are imported, the value is 1 by default. Therefore, the Forced On setting is not recommended unless you are targeting a high-spec device. Forced On can be set from "Anisotropic Textures" in "Project Settings -> Quality".



▲ Figure 4.3 Forced On Settings

Make sure that the Aniso Level setting is not enabled for objects that have no effect, or that it is not set too high for objects that do have an effect.

The effect of Aniso Level is not linear, but rather switches in steps. The author verified that it changes in four steps: 0~1, 2-3, 4~7, and 8 or later.

4.1.5 Compression Settings

Textures should be compressed unless there is a specific reason not to. If you find uncompressed textures in your project, it may be human error or lack of regulation. Check it out as soon as possible. More information on compression settings can be found at "2.3.3 Image Compression".

We recommend using TextureImporter to automate these compression settings to avoid human error.

▼ List 4.2 Example of TextureImporter automation

```
1: using UnityEditor;
2:
3: public class ImporterExample : AssetPostprocessor
4: {
5:     private void OnPreprocessTexture()
6:     {
7:         var importer = assetImporter as TextureImporter;
8:         // Read/Write settings, etc. are also possible.
9:         importer.isReadable = false;
10:
11:         var settings = new TextureImporterPlatformSettings();
12:         // Specify Android = "Android", PC = "Standalone
13:         settings.name = "iPhone";
14:         settings.overridden = true;
15:         settings.textureCompression = TextureImporterCompression.Compressed;
16:         // Specify compression format
17:         settings.format = TextureImporterFormat.ASTC_6x6;
18:         importer.SetPlatformTextureSettings(settings);
19:     }
20: }
```

Not all textures need to be in the same compression format. For example, among UI images, images with overall gradations tend to show a noticeable quality loss due to compression. In such cases, it is recommended to set a lower compression ratio for only some of the target images. On the other hand, for textures such as 3D models, it is difficult to see the quality loss, so it is best to find an appropriate setting such as a high compression ratio.

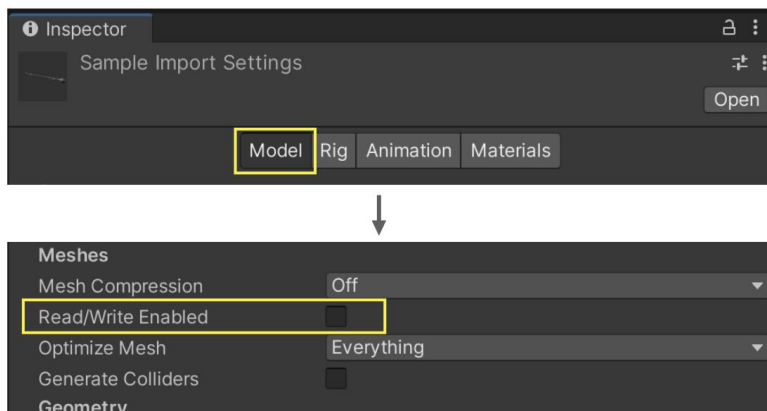
4.2 Mesh

The following are points to keep in mind when dealing with mesh (models) imported into Unity. Performance of imported model data can be improved depending on the settings. The following four points should be noted.

- Read/Write Enabled
- Vertex Compression
- Mesh Compression
- Optimize Mesh Data

4.2.1 Read/Write Enabled

The first mesh note is Read/Write Enabled. This option in the model inspector is disabled by default.



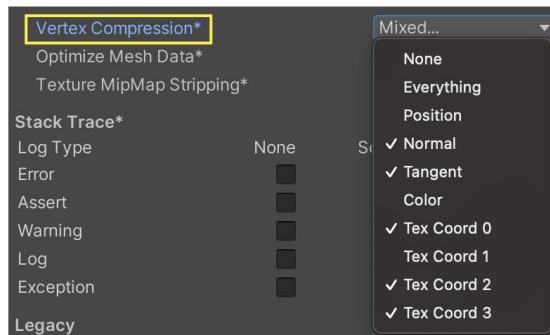
▲ Figure 4.4 Read/Write Settings

If you do not need to access the mesh during runtime, you should disable it. Specifically, if the model is placed on Unity and used only to play an AnimationClip, Read/Write Enabled is fine to disable.

Enabling Read/Write Enabled will consume twice as much memory because information accessible by the CPU is stored in memory. Please check it out, as simply disabling it will save memory.

4.2.2 Vertex Compression

Vertex Compression is an option that changes the precision of mesh vertex information from float to half. This **can reduce memory usage and file size at runtime**. The default setting is as follows.



▲ Figure 4.5 Default settings for Vertex Compression

However, please note that Vertex Compression is disabled under the following conditions

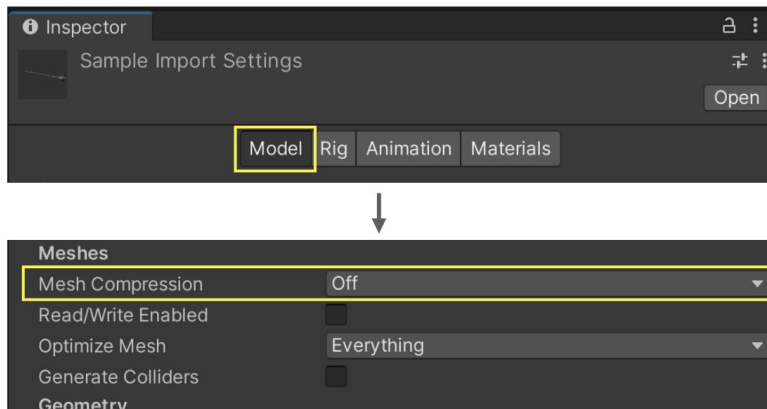
- Read/Write is enabled
- Mesh Compression is enabled
- Mesh with Dynamic Batching enabled and adaptable (less than 300 vertices and less than 900 vertex attributes)

4.2.3 Mesh Compression

Mesh Compression allows you to change the compression ratio of the mesh. The higher the compression ratio, the smaller the file size and the less storage space is required. Compressed data is decompressed at runtime. Therefore, **memory usage at runtime is not affected**.

Mesh Compression offers four compression settings.

- Off: Uncompressed
- Low: Low compression
- Medium: Medium compression
- High: High compression



▲ Figure 4.6 Mesh Compression

As mentioned in "4.2.2 Vertex Compression", enabling this option **disables Vertex Compression**. Especially for projects with strict memory usage limitations, please be aware of this disadvantage before setting this option.

4.2.4 Optimize Mesh Data

Optimize Mesh Data is a function that automatically deletes unnecessary vertex data from the mesh. Unnecessary vertex data is automatically determined based on the shader used. This has the effect of reducing both memory and storage at runtime. The setting can be configured in "Other" under "Project Settings -> Player".



▲ Figure 4.7 Optimize Mesh Data setting

This option is useful because it automatically deletes vertex data, but be aware that it may cause unexpected problems. For example, when switching between Material and Shader at runtime, the properties accessed may be deleted, resulting in incorrect rendering results. When bundling only Mesh assets, the incorrect Material settings may result in unnecessary vertex data. This is common in cases where only a mesh reference is provided, such as in the Particle System.

4.3 Material

Material is an important function that determines how an object is rendered. Although it is a familiar feature, it can easily cause memory leaks if used incorrectly. In this section, we will show you how to use materials safely.

Simply accessing a parameter will duplicate it.

The most important thing to remember about materials is that they can be duplicated simply by accessing their parameters. And it is hard to notice that it is being duplicated.

Take a look at the following code

▼ List 4.3 Example of Material being duplicated

```
1: Material material;  
2:  
3: void Awake()  
4: {  
5:     material = renderer.material;  
6:     material.color = Color.green;  
7: }
```

This is a simple process of setting the material's color property to Color.green. The renderer's material is duplicated. And the duplicated object must be explicitly Destroyed.

▼ List 4.4 Example of deleting a duplicated Material

```
1: Material material;  
2:  
3: void Awake()  
4: {  
5:     material = renderer.material;  
6:     material.color = Color.green;  
7: }  
8:  
9: void OnDestroy()  
10: {  
11:     if (material != null)  
12:     {  
13:         Destroy(material)  
14:     }  
15: }
```

Destroying a duplicated material in this way avoids memory leaks.

Let's thoroughly clean up generated materials.

Dynamically generated materials are another common cause of memory leaks. Make sure to Destroy generated materials when you are done using them.

Take a look at the following sample code.

▼ List 4.5 Example of deleting a dynamically generated material

```
1: Material material;
2:
3: void Awake()
4: {
5:     material = new Material(); // Dynamically generated material
6: }
7:
8: void OnDestroy()
9: {
10:     if (material != null)
11:     {
12:         Destroy(material); // Destroying a material when you have finished using it
13:     }
14: }
```

Materials should be destroyed when they are finished being used (OnDestroy). Destroy materials at the appropriate timing according to the rules and specifications of the project.

4.4 Animation

Animation is a widely used asset in both 2D and 3D. This section introduces practices related to animation clips and animators.

4.4.1 Adjusting the number of skin weights

Internally, motion updates the position of each vertex by calculating how much of each bone is affected by each vertex. The number of bones taken into account in the position calculation is called the skinweight or influence count. Therefore, the load can be reduced by adjusting the number of skin weights. However, reducing the number of skin weights may result in a strange appearance, so be sure to verify this

Chapter 4 Tuning Practice - Asset

when adjusting the number of skin weights.

The number of skin weights can be set from "Other" under "Project Settings -> Quality."



▲ Figure 4.8 Adjusting Skin Weight

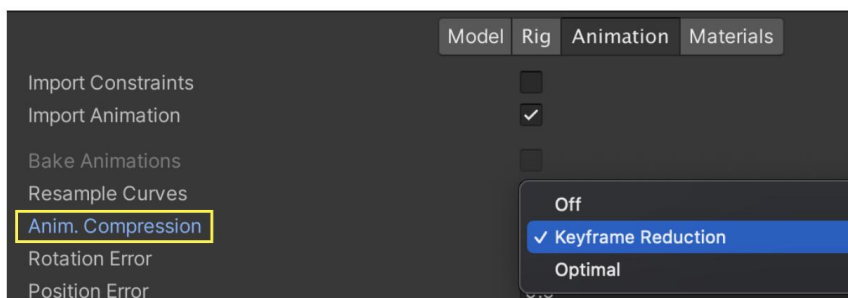
This setting can also be adjusted dynamically from a script. Therefore, it is possible to set Skin Weights to 2 for low-spec devices and 4 for high-spec devices, and so on, for fine-tuning.

▼ List 4.6 Changing SkinWeight settings

```
1: // How to switch QualitySettings entirely
2: // The argument number is the order of the QualitySettings, starting with 0.
3: QualitySettings.SetQualityLevel(0);
4:
5: // How to change only SkinWeights
6: QualitySettings.skinWeights = SkinWeights.TwoBones;
```

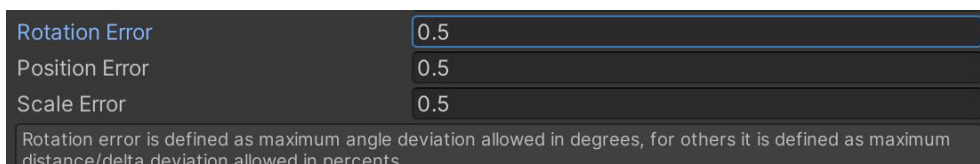
4.4.2 Reducing Keys

Animation files are dependent on the number of keys, which can be a drain on storage and memory at run-time. One way to reduce the number of keys is to use the Anim. Compression feature. This option can be found by selecting the Animation tab from the model import settings. When Anim. Compression is enabled, unnecessary keys are automatically removed during asset import.



▲ Figure 4.9 Anim. Compression settings screen

Keyframe Reduction reduces keys when there is little change in value. Specifically, keys are removed when they are within the Error range compared to the previous curve. This error range can be adjusted.



▲ Figure 4.10 Error Settings

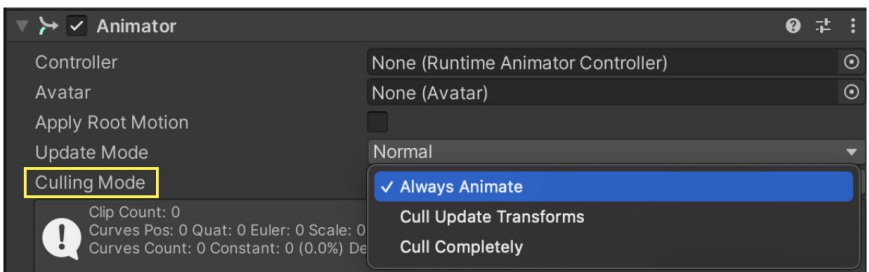
The Error settings are a bit complicated, but the units of the Error settings differ depending on the item. Rotation is in degrees, while Position and Scale are in percent. The tolerance for a captured image is 0.5 degrees for Rotation, and 0.5% for Position and Scale. The detailed algorithm can be found in the Unity documentation at ^{*1}, so please take a peek if you are interested.

Optimal is even more confusing, but it compares two reduction methods, the Dense Curve format and Keyframe Reduction, and uses the one with the smaller data. The key point to keep in mind is that Dense Curve is smaller in size than Keyframe Reduction. However, it tends to be noisy, which may degrade the animation quality. After understanding this characteristic, let's visually check the actual animation to see if it is acceptable.

^{*1} <https://docs.unity3d.com/Manual/class-AnimationClip.html#tolerance>

4.4.3 Reduction of update frequency

By default, Animator updates every frame even if the animation is not on screen. There is an option called Culling Mode that allows you to change this update method.



▲ Figure 4.11 Culling Mode

The meaning of each option is as follows

▼ Table 4.1 Description of culling mode

Type	Meaning
Always Animate	Always update even when off-screen. (Default setting)
Cull Update Transform	Do not write IK or Transform when off-screen. State machine updates are performed.
Cull Completely	No state machine updates are performed when off-screen. Animation stops completely.

There are a few things to note about each option. First, be careful when using Root motion when setting Cull Completely. For example, if you have an animation that frames in from off-screen, the animation will stop immediately because it is off-screen. As a result, the animation will not frame in forever.

Next is Cull Update Transform. This seems like a very useful option, since it only skips updating the transform. However, be careful if you have a shaking or other Transform-dependent process. For example, if a character goes out of frame, no updates will be made from the pose at that time. When the character enters the frame again, it will be updated to a new pose, which may cause the shaking object to move significantly. It is a good idea to understand the pros and cons of each option before changing the settings.

Also, even with these settings, it is not possible to dynamically change the frequency of animation updates in detail. For example, you can optimize the frequency of animation updates by halving the frequency of animation updates for objects that are farther away from the camera. In this case, you need to use `AnimationClipPlayable` or deactivate `Animator` and call `Animator.Update` yourself. Both require writing your own scripts, but the latter is easier to implement than the former.

4.5 Particle System

Game effects are essential for game presentation, and Unity often uses the Particle System. In this chapter, we will introduce how to use the Particle System from the perspective of performance tuning and what to keep in mind to avoid mistakes.

The following two points are important.

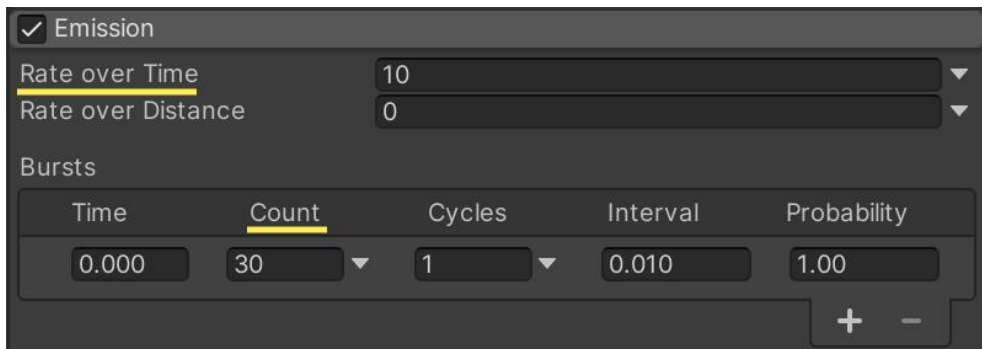
- Keep the number of particles low.
- Be aware that noise is heavy.

4.5.1 Reduce the number of particles

The number of particles is related to the load, and since the Particle System is CPU-powered (CPU particles), the more particles there are, the higher the CPU load. As a basic policy, set the number of particles to the minimum necessary. Adjust the number of particles as needed.

There are two ways to limit the number of particles.

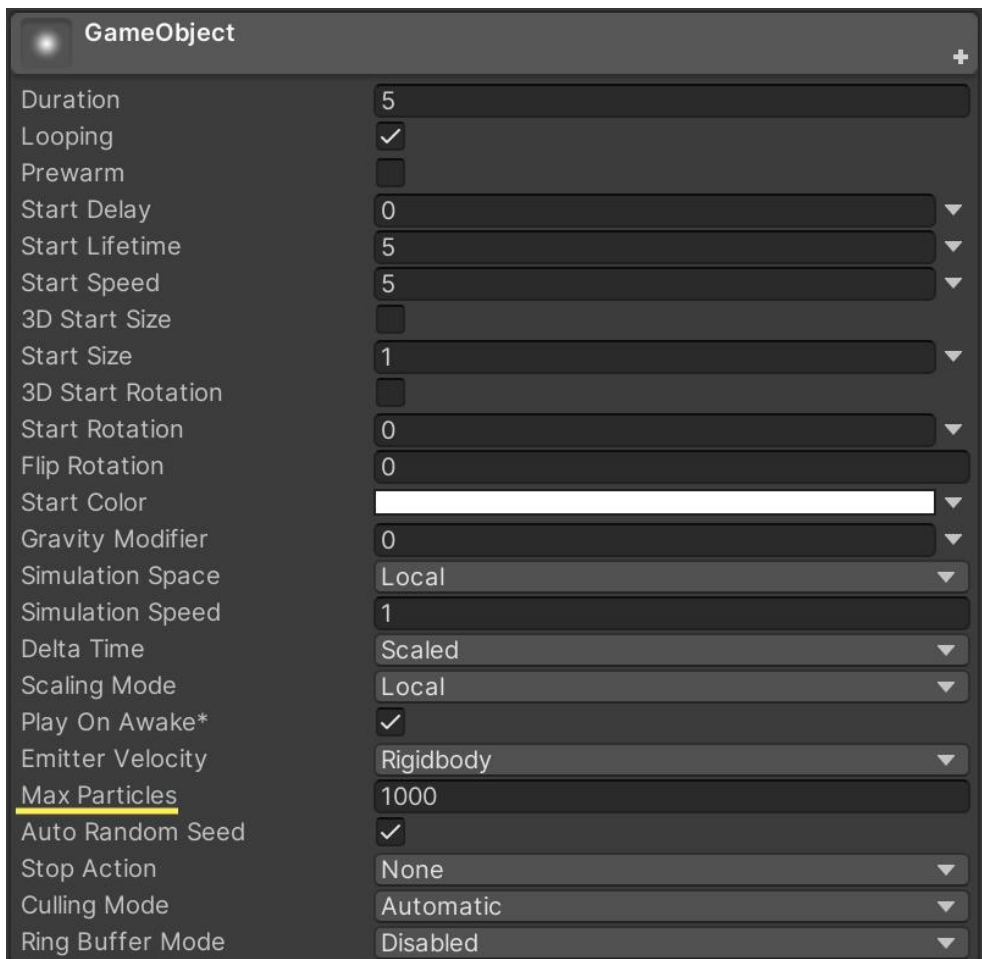
- Limit the number of particles emitted by the Emission module
- Limit the maximum number of particles in the Max Particles main module



▲ Figure 4.12 Limit on the number of emissions in the Emission module

- Rate over Time: Number of parts emitted per second
- Bursts > Count: Number of particles to be emitted at burst timing

Adjust these settings to achieve the minimum number of particles required.

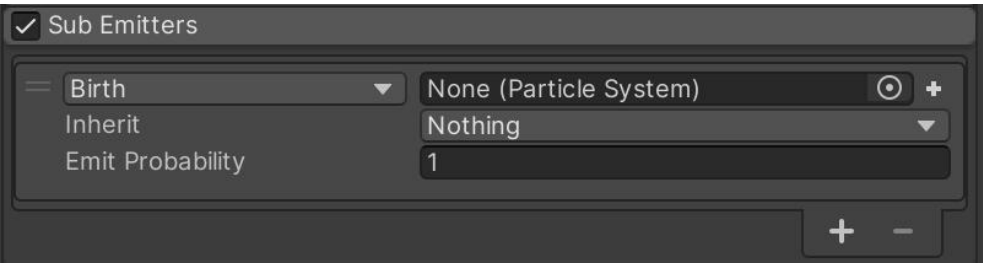


▲ Figure 4.13 Limiting the number of particles emitted with Max Particles

Another way is to use Max Particles in the main module. In the example above, particles over 1000 will not be emitted.

Be careful with Sub Emitters

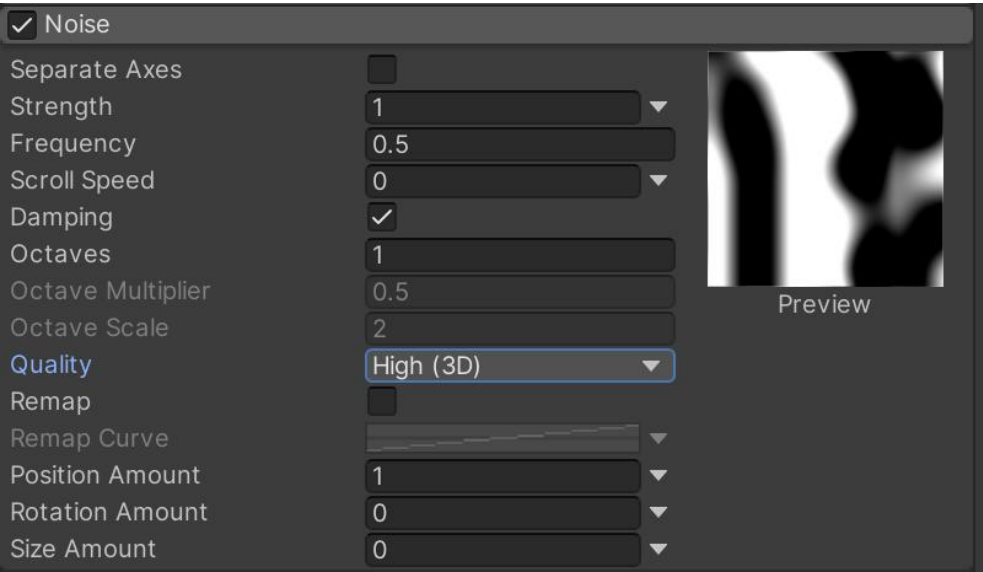
The Sub Emitters module should also be considered when reducing the number of particles.



▲ Figure 4.14 Sub Emitters Module

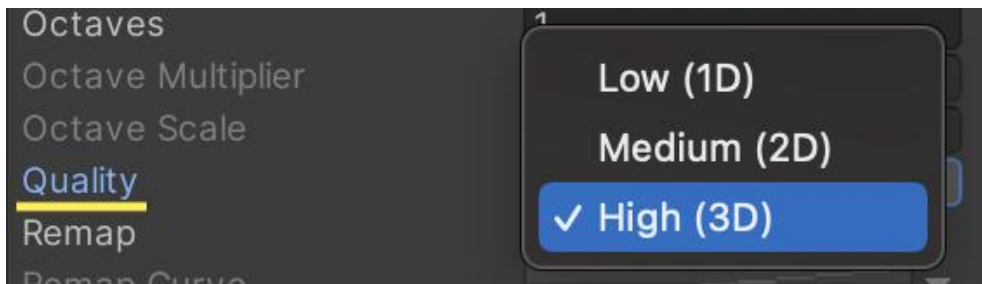
The Sub Emitters module generates arbitrary particle systems at specific times (at creation, at the end of life, etc.) Depending on the Sub Emitters settings, the number of particles may reach the peak number at once, so be careful when using this module.

4.5.2 Note that noise is heavy.



▲ Figure 4.15 Noise Module

The Noise module's Quality is easily overloaded. Noise can express organic particles and is often used to easily increase the quality of effects. Because it is a frequently used function, you should be careful about its performance.



▲ Figure 4.16 Quality of the Noise module

- Low (1D)
- Midium (2D)
- High (3D)

The higher the dimension of Quality, the higher the load. If you do not need Noise, turn off the Noise module. If you need to use noise, set the Quality setting to Low first, and then increase the Quality according to your requirements.

4.6 Audio

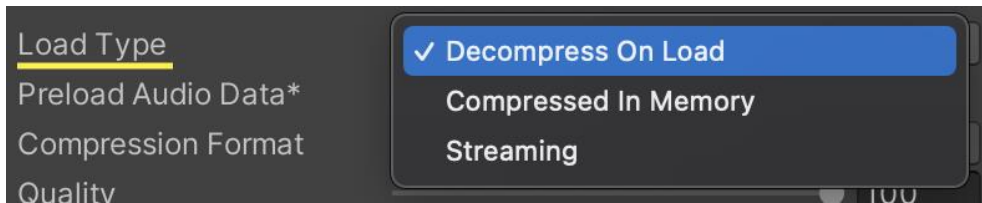
The default state with sound files imported has some improvement points in terms of performance. The following three settings are available.

- Load Type
- Compression Format
- Force To Mono

Set these settings appropriately for background music, sound effects, and voices that are often used in game development.

4.6.1 Load Type

There are three types of ways to load a sound file (AudioClip).



▲ Figure 4.17 AudioClip LoadType

- Decompress On Load
- Compressed In Memory
- Streaming

Decompress On Load

Decompress On Load loads uncompressed video into memory. It is less CPU-intensive, so playback is performed with less wait time. On the other hand, it uses a lot of memory.

It is recommended for short sound effects that require immediate playback. BGM and long voice files use a lot of memory, so care should be taken when using this function.

Compressed In Memory

Compressed In Memory loads an AudioClip into memory in a compressed state. This means that it is decompressed at the time of playback. This means that the CPU load is high and playback delays are likely to occur.

It is suitable for sounds with large file sizes that you do not want to decompress directly into memory, or for sounds that do not suffer from a slight playback delay. It is often used for voice.

Streaming

Streaming, as the name implies, is a method of loading and playing back sounds. It uses less memory, but is more CPU-intensive. It is recommended for use with long BGM.

▼ Table 4.2 Summary of Loading Methods and Main Uses

Type	Usage
Decompress On Load	Sound Effects
Compressed In Memory	Voice
Streaming	BGM

4.6.2 Compression Format

Compression format is the compression format of the AudioClip itself.



▲ Figure 4.18 AudioClip Compression Format

PCM

Uncompressed and consumes a large amount of memory. Do not set this unless you want the best sound quality.

ADPCM

Uses 70% less memory than PCM, but the quality is lower, and the CPU load is much smaller than with Vorbis. The CPU load is much lower than Vorbis, which means that the speed of decompression is faster, making it suitable for immediate playback and for sounds that are played back in large quantities. This is especially true for noisy sounds such as footsteps, collisions, weapons, etc., that need to be played back quickly and in large quantities.

Vorbis

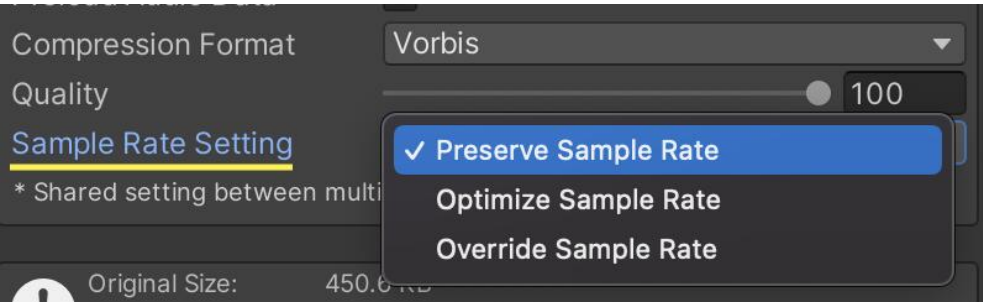
As a lossy compression format, the quality is lower than PCM, but the file size is smaller. It is the only format that allows for fine-tuning of the sound quality. It is the most used compression format for all sounds (background music, sound effects, voices).

▼ Table 4.3 Summary of Compression Methods and Main Uses

Type	Usage
PCM	Not used
ADPCM	Sound Effects
Vorbis	BGM, sound effects, voice

4.6.3 Sample Rate

Quality can be adjusted by specifying the sample rate. All compressed formats are supported. Three methods can be selected from Sample Rate Setting.



▲ Figure 4.19 Sample Rate Settings

Preserve Sample Rate

Default setting. The sample rate of the original sound source is used.

Optimize Sample Rate

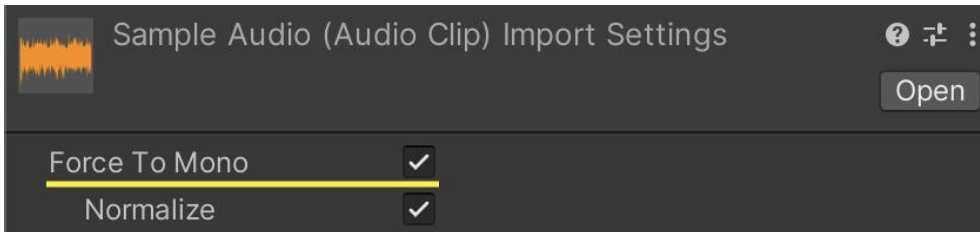
Analyzed by Unity and automatically optimized based on the highest frequency component.

Override Sample Rate

Override the sample rate of the original sound source. 8,000 to 192,000 Hz can be specified. The quality will not be improved even if the sample rate is higher than the original source. Use this option when you want to lower the sample rate than the original sound source.

4.6.4 Set Force To Mono for sound effects.

By default, Unity plays stereo, but by enabling Force To Mono, mono playback is enabled. Enabling mono playback will cut the file size and memory size in half, since there is no need to have separate data for left and right channels.



▲ Figure 4.20 AudioClip Force To Mono

Mono playback is often fine for sound effects. In some cases, mono playback is also better for 3D sound. It is recommended to enable Force To Mono after careful consideration. The performance tuning effect is a mountain out of a molehill. If you have no problem with monaural playback, you should actively use Force To Mono.

Although this is not the same as performance tuning, uncompressed audio files should be imported into Unity. If you import compressed audio files, they will be decoded and recompressed on the Unity side, resulting in a loss of quality.

4.7 Resources / StreamingAssets

There are special folders in the project. The following two in particular require attention from a performance standpoint.

- Resources folder
- StreamingAssets folder

Normally, Unity only includes objects referenced by scenes, materials, scripts, etc.

in a build.

▼ List 4.7 Example of an object referenced by a script

```
1: // Referenced objects are included in the build
2: [SerializeField] GameObject sample;
```

The rules are different for the special folders mentioned above. Stored files are included in the build. This means that even files that are not actually needed are included in the build if they are stored, leading to an expansion of the build size.

The problem is that it is not possible to check from the program. You have to visually check for unnecessary files, which is time consuming. Be careful adding files to these folders.

However, the number of stored files will inevitably increase as the project progresses. Some of these files may be mixed in with unnecessary files that are no longer in use. In conclusion, we recommend that you review your stored files on a regular basis.

4.7.1 Resources folder slows down startup time

Storing a large number of objects in the Resources folder will increase application startup time. The Resources folder is an old-fashioned convenience feature that allows you to load objects by string reference.

▼ List 4.8 Example of an object referenced in a script

```
1: var object = Resources.Load("aa/bb/cc/obj");
```

You can load objects with code like this. It is easy to overuse the Resources folder because you can access objects from scripts by storing them in the Resources folder. However, overloading the Resources folder will increase the startup time of the application, as mentioned above. The reason for this is that when Unity starts up, it analyzes the structure in all Resources folders and creates a lookup table. It is best to minimize the use of the Resources folder as much as possible.

4.8 ScriptableObject

ScriptableObjects are YAML assets, and many projects are likely to manage their files as text files. By explicitly specifying an `[PreferBinarySerialization]` Attribute to change the storage format to binary. For assets that are mainly large amounts of data, binary format improves the performance of write and read operations.

However, binary format is naturally more difficult to use with merge tools. For assets that need to be updated only by overwriting the asset, such as those for which there is no need to check the text for changes (), or for assets whose data is no longer being changed after game development is complete, it is recommended to use `[PreferBinarySerialization]` ScriptableObjects are not required to be used with ScriptableObjects.

A common mistake when using ScriptableObjects is mismatching class names and source code file names. The class and file must have the same name. Be careful with naming when creating classes and make sure that the `.asset` file is correctly serialized and saved in the binary format.

▼ List 4.9 Example Implementation of ScriptableObject

```
1: /*
2: * When the source code file is named ScriptableObjectSample.cs
3: */
4:
5: // Serialization succeeded
6: [PreferBinarySerialization]
7: public sealed class ScriptableObjectSample : ScriptableObject
8: {
9:     ...
10: }
11:
12: // Serialization Failure
13: [PreferBinarySerialization]
14: public sealed class MyScriptableObject : ScriptableObject
15: {
16:     ...
17: }
```



PERFORMANCE TUNING BIBLE

CHAPTER

05

第5章

**Tuning Practice
— AssetBundle —**

CyberAgent Smartphone Games & Entertainment

Chapter 5

Tuning Practice - AssetBundle

Problems in AssetBundle configuration can cause many problems, such as wasting valuable communication and storage space for the user, as well as hindering the comfortable game play. This chapter describes the configuration and implementation policies for AssetBundle.

5.1 Granularity of AssetBundle

The granularity of the AssetBundle should be carefully considered due to dependency issues. At the extreme, there are two ways to do this: put all assets in one AssetBundle, or put each asset in one AssetBundle. Both methods are simple, but the former method has a critical problem. The former method is simple, but the former method has a fatal problem: even if you only add assets or update one asset, you have to recreate the entire file and distribute it. If the total amount of assets is in GB, the update load is very high.

Therefore, the method of dividing the AssetBundle as much as possible is chosen, but if it is too detailed, it will cause overhead in various areas. Therefore, we basically recommend the following policy for making AssetBundle.

- Assets that are supposed to be used at the same time should be combined into a single AssetBundle.
- Assets that are referenced by multiple assets should be in separate AssetBundles.

It is difficult to control perfectly, but it is a good idea to set some rules regarding granularity within the project.

5.2 Load API for AssetBundle

There are three types of APIs for loading assets from AssetBundle.

`AssetBundle.LoadFromFile`

Load by specifying the file path that exists in storage. This is usually used because it is the fastest and most memory-efficient.

`AssetBundle.LoadFromMemory`

Load by specifying the `AssetBundle` data already loaded in memory. While using `AssetBundle`, a very large amount of data needs to be maintained in memory, and the memory load is very large. For this reason, it is not normally used.

`AssetBundle.LoadFromStream`

Load by specifying `Stream` which returns the `AssetBundle` data. When loading an encrypted `AssetBundle` while decrypting it, use this API in consideration of the memory load. However, since `Stream` must be seekable, be careful not to use a cipher algorithm that cannot handle seek.

5.3 AssetBundle unloading strategy

If `AssetBundle` is not unloaded when it is no longer needed, it will overwhelm memory. The argument `unloadAllLoadedObjects` of `AssetBundle.Unload(bool unloadAllLoadedObjects)`, which is the API to be used in this case, is very important and should be decided how to set it up at the beginning of the development. If this argument is true, when unloading an `AssetBundle`, all assets loaded from that `AssetBundle` will also be unloaded. If false, no assets are unloaded.

In other words, the true case, which requires the `AssetBundle` to be loaded continuously while the assets are being used, is more memory-intensive, but it is also safer because it ensures that the assets are destroyed. On the other hand, the false case has a low memory load because the `AssetBundle` can be unloaded when the asset is finished loading. However, forgetting to unload the used assets can lead to memory leaks or cause the same asset to be loaded multiple times in memory, so proper memory management is required. Therefore, proper memory management is required. In general, strict memory management is severe, so `AssetBundle.Unload(true)` is recommended if memory load is sufficient.

5.4 Optimization of the number of simultaneously loaded AssetBundles

In the case of `AssetBundle.Unload(true)`, `AssetBundle` cannot be unloaded while assets are in use. Therefore, depending on the granularity of the `AssetBundle`, there may be situations where more than 100 `AssetBundles` are loaded at the same time. In this case, you need to be careful about the file descriptor limit and the memory usage of `PersistentManager.Remapper`.

A file descriptor is an operation ID assigned by the OS when reading or writing a file. One file descriptor is required to read or write one file, and the file descriptor is released when the file operation is completed. Since there is a limit to the number of file descriptors a process can have, it is not possible to have more than this number of files open at the same time. If you see the error message "Too many open files", it means that the process has reached the limit. Therefore, the number of simultaneous loads in the `AssetBundle` is affected by this limit, and Unity also has to keep a certain amount of margin for the limit, since it has to open some files. This limit varies depending on the OS and version, so it is necessary to investigate the value for the target platform in advance. Even if the limit is hit, it is possible to temporarily raise the limit depending on the OS^{*1}, so consider implementing this if necessary.

A second problem with having many `AssetBundles` to load at the same time is the presence of `PersistentManager.Remapper` in Unity. Simply put, the `PersistentManager` is a function that manages the mapping relationship between objects and data within Unity. In other words, you can imagine that it uses memory in proportion to the number of `AssetBundles` loaded at the same time, but the problem is that even if you release an `AssetBundle`, the memory space used is not released, but pooled. Because of this nature, memory will be squeezed in proportion to the number of concurrent loads, so it is important to reduce the number of concurrent loads.

Based on the above, when operating under the `AssetBundle.Unload(true)` policy, it is recommended that the maximum number of concurrently loaded `AssetBundles` be 150 to 200, and when operating under the `AssetBundle.Unload(false)` policy, it is recommended that the maximum number be 150 or less.

^{*1} In Linux/Unix environments, the limit can be changed at runtime using the `setrlimit` function



PERFORMANCE TUNING BIBLE

CHAPTER 06

第6章

**Tuning Practice
— Physics —**

CyberAgent Smartphone Games & Entertainment

Chapter 6

Tuning Practice - Physics

This chapter introduces Physics optimization.

Physics here refers to physics operations using PhysX, not ECS's Unity Physics.

This chapter focuses mainly on 3D Physics, but 2D Physics may also be useful in many areas.

6.1 Turning Physics On and Off

By Unity standard, even if there is no physics component in the scene, the physics engine will always perform physics calculations in every frame. Therefore, if you do not need physics in your game, you should turn off the physics engine.

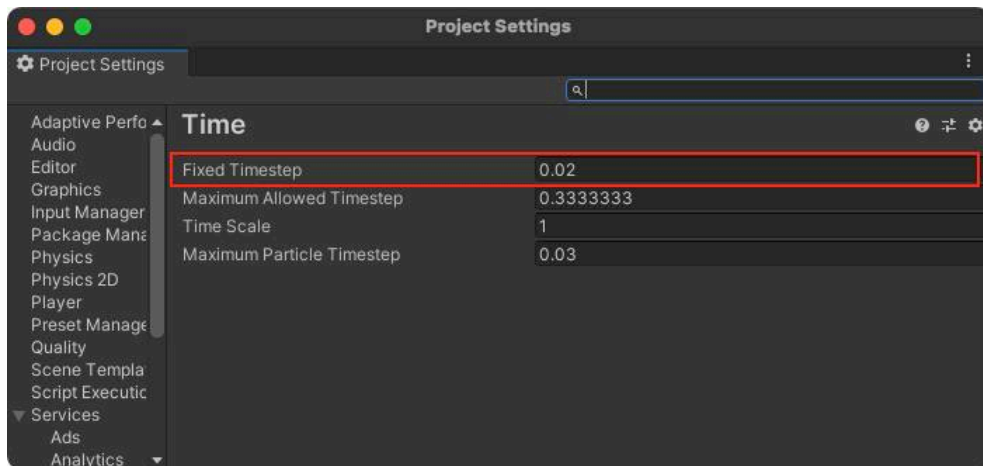
Physics engine processing can be turned on or off by setting a value to `Physics.autoSimulation`. For example, if you want to use physics only ingame and not otherwise, set this value to `true` only ingame.

6.2 Optimizing Fixed Timestep and Fixed Update Frequency

`MonoBehaviour's FixedUpdate` runs at a fixed time, unlike `Update`.

The physics engine calls `Fixed Update` multiple times in one frame to match the elapsed time in the game world with the time in the physics engine world. Therefore, the smaller the value of `Fixed Timestep`, the more **times Fixed Update is called, which causes load**.

This time is set in Project Settings as shown at Figure 6.1. **Fixed Timestep** in Project Settings as shown at . The unit for this value is seconds. The units for this value are seconds. The default value is 0.02, or 20 milliseconds.



▲ Figure 6.1 Fixed Timestep item in Project Settings

It can also be changed from within the script by manipulating `Time.fixedDeltaTime`.

Fixed Timestep is generally smaller, the more accurate the physics calculations are and the less likely it is that problems such as collision loss will occur. Therefore, although it is a tradeoff between accuracy and load, it is desirable to **set this value as close to the target FPS as possible without causing game behavior problems**.

6.2.1 Maximum Allowed Timestep

As mentioned in the previous section, Fixed Update is called multiple times based on the elapsed time from the previous frame.

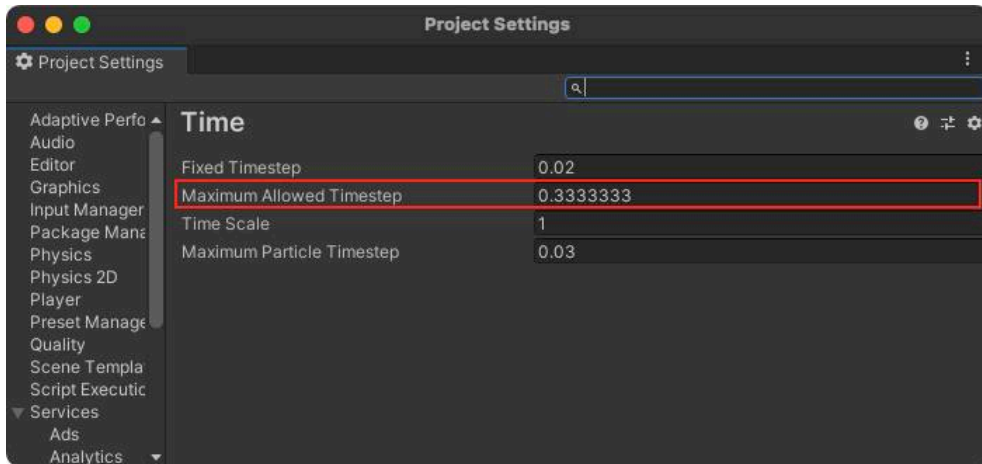
If the elapsed time from the previous frame is large, for example, due to heavy rendering in a certain frame, Fixed Update will be called more often than usual in that frame.

For example, if **Fixed Timestep** is 20 milliseconds and the previous frame took 200 milliseconds, Fixed Update will be called 10 times.

This means that if one frame is dropped, the cost of physics operations in the next frame will be higher. This increases the risk that the frame will also fail, which in turn makes the physics operations in the next frame heavier, a phenomenon known in the physics engine world as a negative spiral.

Figure 6.2 To solve this problem, Unity allows the user to set the **Maximum Al-**

lowed Timestep which is the maximum amount of time that physics operations can use in a single frame. This value defaults to 0.33 seconds, but you may want to set it closer to the target FPS to limit the number of Fixed Update calls and stabilize the frame rate.



▲ Figure 6.2 Maximum Allowed Timestep item in Project Settings

6.3 Collision Shape Selection

The processing cost of collision detection depends on the shape of the collision and its situation. Although it is difficult to say exactly how much it will cost, a good rule of thumb to remember is that the following collision types are in order of decreasing cost: sphere collider, capsule collider, box collider, and mesh collider.

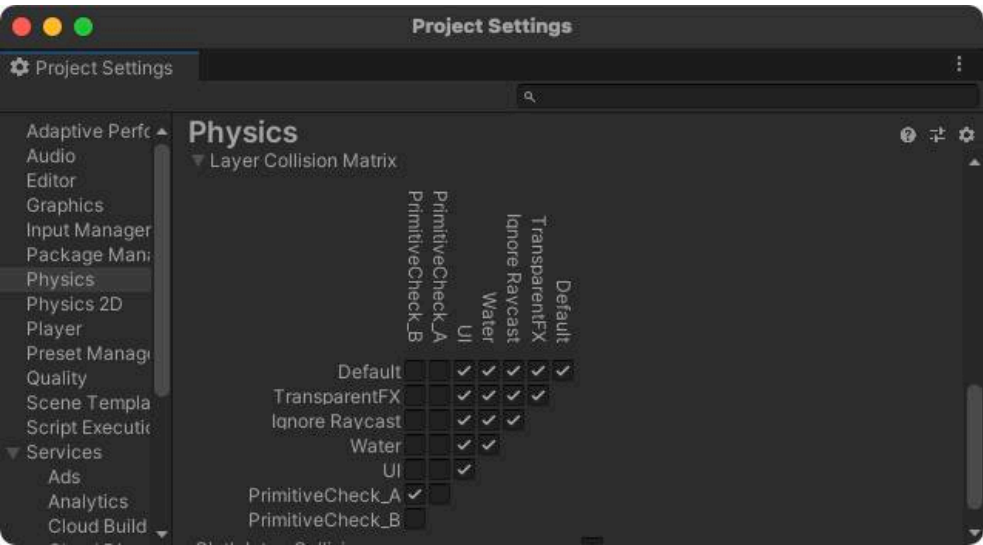
For example, the capsule collider is often used to approximate the shape of a humanoid character, but if height is not a factor in the game specifications, replacing it with a sphere collider will result in a smaller cost of judging a hit.

Also note that among these shapes, the mesh collider is particularly loaded.

First, consider whether a sphere collider, capsule collider, or box collider and its combinations can be used to prepare collisions. If this is still inconvenient, use a mesh collider.

6.4 Collision Matrix and Layer Optimization

Physics has a setting called "collision matrix" that defines which layers of game objects can collide with each other. This setting can be changed from Physics > Layer Collision Matrix in Project Settings as shown at Figure 6.3.



▲ Figure 6.3 Layer Collision Matrix item in Project Settings

The Collision Matrix indicates that if the checkboxes at the intersection of two layers are checked, those layers will collide.

Properly performing this setting is the **most efficient way to eliminate calculations between objects that do not need to collide**, since layers that do not collide are **also excluded from the pre-calculation that takes a rough hit on the object, called the broad phase**.

For performance considerations, it is preferable to have a **dedicated layer for physics calculations** and **uncheck all checkboxes between layers** that do not need to collide.

6.5 Raycast Optimization

Raycasting is a useful feature that allows you to get collision information between rays you fly and colliding colliders, but it can also be a source of load.

6.5.1 Types of Raycasts

In addition to `Physics.Raycast`, which determines collision with a line segment, there are other methods such as `Physics.SphereCast`, which determines collision with other shapes.

However, the more complex the shape to be judged, the higher the load. Considering performance, it is advisable to use only `Physics.Raycast` as much as possible.

6.5.2 Optimization of Raycast Parameters

In addition to the two parameters of starting point and direction of raycast, `Physics.Raycast` has two other parameters for performance optimization: `maxDistance` and `layerMask`.

`maxDistance` specifies the maximum length of the ray cast decision, i.e., the length of the ray.

If this parameter is omitted, `Mathf.Infinity` is passed as the default value, and an attempt is made to take a decision on a very long ray. Such a ray may have a negative impact on the broad phase, or it may hit objects that do not need to be hit in the first place, so do not specify a distance greater than necessary.

`layerMask` also avoids setting up bits in layers that do not need to be hit.

As with the collision matrix, layers with no standing bits are also excluded from the broad phase, thus reducing computational cost. If this parameter is omitted, the default value is `Physics.DefaultRaycastLayers`, which collides with all layers except `Ignore Raycast`, so be sure to specify this parameter as well.

6.5.3 RaycastAll and RaycastNonAlloc

`Physics.Raycast` returns collision information for one of the colliding colliders, but the `Physics.RaycastAll` method can be used to obtain multiple collision information.

`Physics.RaycastAll` returns collision information by dynamically allocating an

Chapter 6 Tuning Practice - Physics

array of `RaycastHit` structures. Therefore, each call to this method will result in a GC Alloc, which can cause spikes due to GC.

To avoid this problem, there is a method called `Physics.RaycastNonAlloc` that, when passed an allocated array as an argument, writes the result to that array and returns it.

For performance considerations, GC Alloc should not occur within `FixedUpdate` whenever possible.

List 6.1 As shown in Figure 2.1, GC.Alloc can be avoided except during array initialization by maintaining the array to which the results are written in a class field, pooling, or other mechanism, and passing that array to `Physics.RaycastNonAlloc`.

▼ List 6.1 `Physics.RaycastAllNonAlloc` Usage of

```
1: // Starting point to skip ray
2: var origin = transform.origin;
3: // Direction of ray
4: var direction = Vector3.forward;
5: // Length of ray
6: var maxDistance = 3.0f;
7: // The layer with which the ray will collide
8: var layerMask = 1 << LayerMask.NameToLayer("Player");
9:
10: // An array to store the ray-cast collision results
11: // This array can be allocated in advance during initialization or
12: // or use the one allocated in the pool.
13: // The maximum number of ray-cast results must be determined in advance
14: // private const int kMaxResultCount = 100;
15: // private readonly RaycastHit[] _results = new RaycastHit[kMaxResultCount];
16:
17: // All collision information is returned in an array.
18: // Return value is number of collisions
19: var hitCount = Physics.RaycastNonAlloc(
20:     origin,
21:     direction,
22:     _results,
23:     layerMask,
24:     query
25: );
26: if (hitCount > 0)
27: {
28:     Debug.Log($"{hitCount}人のプレイヤーとの衝突しました");
29:
30:     // The _results array stores collision information in order.
31:     var firstHit = _results[0];
32:
33:     // Note that indexes exceeding the number of collisions are invalid information.
34: }
```


6.6 Collider and Rigidbody

Unity Physics has two components: `Collider`, which deals with collisions such as sphere colliders and mesh colliders, and `Rigidbody`, which is used for rigidbody-based physics simulations. Depending on the combination of these components and their settings, they are classified into three colliders.

An object to which the `Collider` component is attached and to which the `Rigidbody` component is not attached is called a **Static Collider** (Static Collider).

This collider is optimized to **be used only for geometry that always stays in the same place and never moves**.

Therefore, you **should not enable or disable** the static collider, **nor should you move or scale it** during game play. Doing so will **cause recalculation due to changes in the internal data structures, which can significantly degrade performance**.

Objects to which both the `Collider` and `Rigidbody` components are attached should not be used as **Dynamic Collider** (Dynamic Collider).

This collider can be collided with other objects by the physics engine. It can also react to collisions and forces applied by manipulating the `Rigidbody` component from scripts.

This makes it the most commonly used collider in games that require physics.

A component with both the `Collider` and `Rigidbody` components attached and with the `isKinematic` property of `Rigidbody` enabled is a **Kinematic Dynamic Collider** (A kinematic dynamic collider is a collider that is attached to a component).

Kinematic dynamic colliders can be moved by directly manipulating the `Transform` component, but not by applying collisions or forces by manipulating the `Rigidbody` component like normal dynamic colliders.

This collider can be used to optimize the physics when you **want to switch the execution of physics** operations, or for **obstacles such as** doors that you want to **move occasionally but not the majority of the time**.

6.6.1 Rigidbody and sleep states

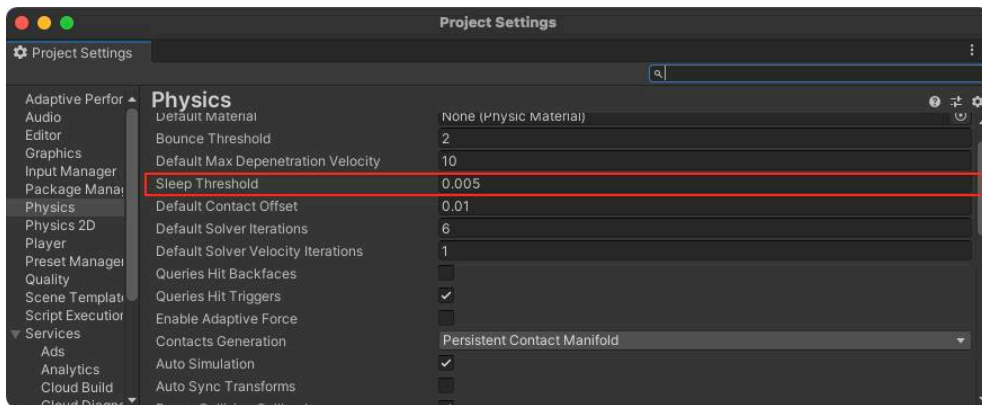
As part of the optimization, the physics engine determines that if an object to which the `Rigidbody` component is attached does not move for a certain period of time, the

Chapter 6 Tuning Practice - Physics

object is considered dormant and its internal state is changed to sleep. Moving to the sleep state minimizes the computational cost for that object unless it is moved by an external force, collision, or other event.

Therefore, objects to which the `Rigidbody` component is attached that do not need to move can be transitioned to the sleep state whenever possible to reduce the computational cost of physics calculations.

`Rigidbody` The threshold used to determine if a component should go to sleep is set in Physics in Project Settings as shown in Figure 6.4. **Sleep Threshold** Sleep Threshold inside Physics in Project Settings, as shown at . Alternatively, if you wish to specify the threshold for individual objects, you can set it from the `Rigidbody.sleepThreshold` property.



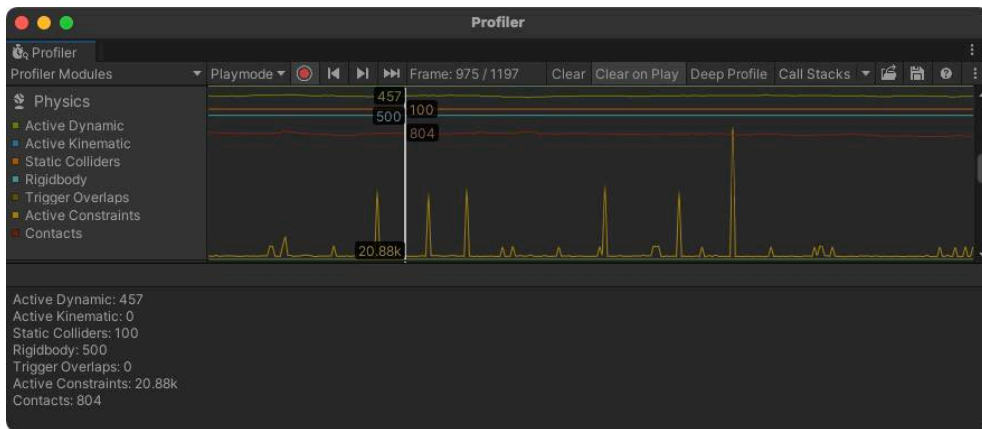
▲ Figure 6.4 Sleep Threshold item in Project Settings

Sleep Threshold represents the mass-normalized kinetic energy of the object when it goes to sleep.

The larger this value is, the faster the object will go to sleep, thus reducing the computational cost. However, the object may appear to come to an abrupt stop because it tends to go to sleep even when moving slowly. If this value is reduced, the above phenomenon is less likely to occur, but on the other hand, it is more difficult for the object to go to sleep, so the computation cost tends to be lower.

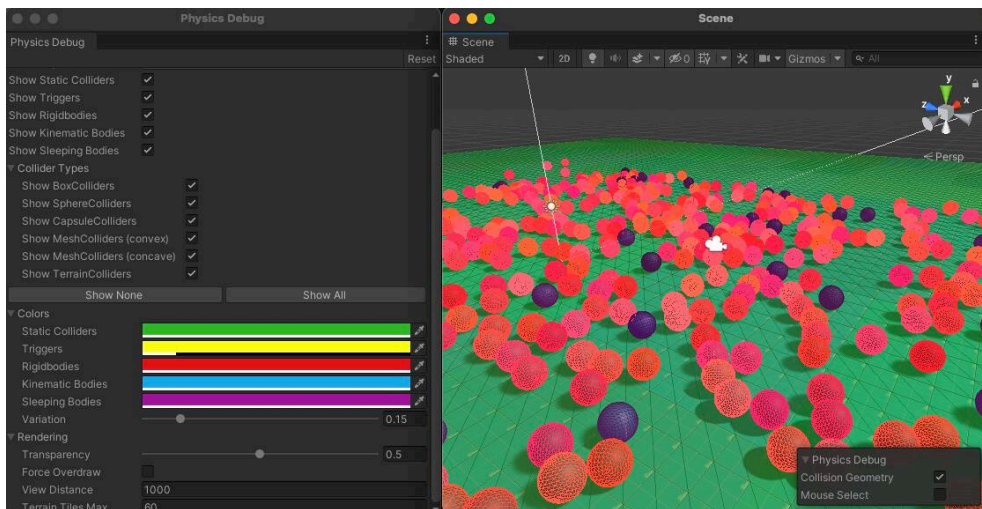
Whether `Rigidbody` is in sleep mode or not can be checked with the `Rigidbody.IsSleeping` property. The total number of `Rigidbody` components active on the scene can be checked from the Physics item in the profiler, as shown at Figure 6.5.

6.6 Collider and Rigidbody



▲ Figure 6.5 Physics item in the Profiler. You can see the number of Rigidbody active as well as the number of each element on the physics engine.

You can also check the number of elements in the **Physics Debugger** to see which objects on the scene are active.



▲ Figure 6.6 Physics Debugger, which displays the state of the objects on the scene in terms of the physics engine, color-coded.

6.7 Collision Detection Optimization

The `Rigidbody` component allows you to select the algorithm to be used for collision detection in the Collision Detection item.

As of Unity 2020.3, there are four collision detection options

- Discrete
- Continuous
- Continuous Dynamic
- Continuous Speculative

These algorithms can be broadly divided into **Discrete Collision Determination** and **Continuous Speculative** and Continuous Speculative **Discrete** is discrete collision detection and the others belong to continuous collision detection.

Discrete collision detection, as the name implies, teleports objects discretely for each simulation, and collision detection is performed after all objects have been moved. Therefore, there is a **possibility of missing a collision**, especially if the objects **are moving at high speed, causing the objects to slip through**.

Continuous collision detection, on the other hand, takes into account collisions between objects before and after they move, thus **preventing fast-moving objects from slipping through**. The computational cost is therefore higher than for discrete collision detection. To optimize performance, **create game behavior so that Discrete can be selected** whenever possible.

If this is inconvenient, consider Continuous collision detection. **Continuous** can be used for **Dynamic Collider** and **Static Collider** and only for the combination of **Continuous Dynamic** enables continuous collision detection even for dynamic colliders. The computational cost is higher for Continuous Dynamic.

Therefore, if you only want to consider collision detection between dynamic and static colliders, i.e., if your character runs around the field, choose Continuous Dynamic.

Continuous Speculative is less computationally expensive than Continuous Dynamic, despite the fact that continuous speculative collisions between dynamic colliders are valid, but it is also less expensive than Continuous Dynamic for ghost collisions (**Ghost Collision** However, it should be introduced with caution because of a phenomenon called "Ghost Collision," which occurs when multiple colliders collide

with each other in close proximity.

6.8 Optimization of other project settings

In addition to the settings introduced so far, here are some other project settings that have a particular impact on performance optimization.

6.8.1 `Physics.autoSyncTransforms`

In versions prior to Unity 2018.3, the position of the physics engine was automatically synchronized with `Transform` each time an API for physics operations such as `Physics.Raycast` was called.

This process is relatively heavy and can cause spikes when calling APIs for physics operations.

To work around this issue, a setting called `Physics.autoSyncTransforms` has been added since Unity 2018.3. Setting this value to `false` will prevent the `Transform` synchronization process described above when calling the physics API.

Synchronization of `Transform` will be performed after `FixedUpdate` is called during physics simulation. This means that if you move the collider and then perform a raycast on the new position of the collider, the raycast will not hit the collider.

6.8.2 `Physics.reuseCollisionCallbacks`

Prior to Unity 2018.3, every time an event was called to receive a collision call for a `Collider` component such as `OnCollisionEnter`, a new `Collision` instance of the argument was created and passed, resulting in a GC Alloc.

Since this behavior can have a negative impact on game performance depending on how often events are called, a new property `Physics.reuseCollisionCallbacks` has been exposed since 2018.3.

Setting this value to `true` will suppress GC Alloc as it internally uses the `Collision` instance that is passed around when calling events.

This setting has a default value of `true` in 2018.3 and later, which is fine if you created your project with a relatively new Unity version, but if you created your project with a version prior to 2018.3, this value may be set to `false`. If this **setting** is disabled, **you should enable it and then modify your code so that the game runs correctly.**



PERFORMANCE TUNING BIBLE

CHAPTER

07

第7章

**Tuning Practice
— Graphics —**

CyberAgent Smartphone Games & Entertainment

Chapter 7

Tuning Practice - Graphics

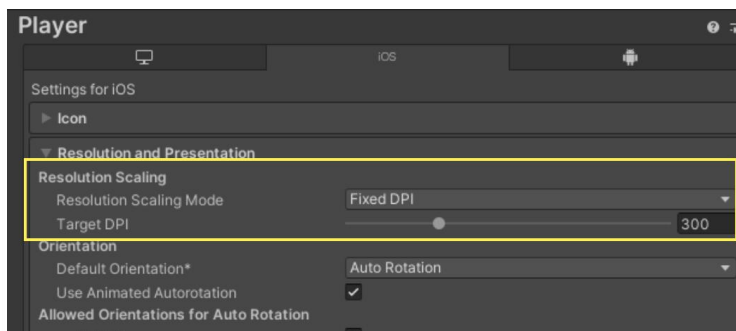
This chapter introduces tuning practices around Unity's graphics capabilities.

7.1 Resolution Tuning

In the rendering pipeline, the cost of fragment shaders increases in proportion to the resolution at which they are rendered. Especially with the high display resolutions of today's mobile devices, it is necessary to adjust the rendering resolution to an appropriate value.

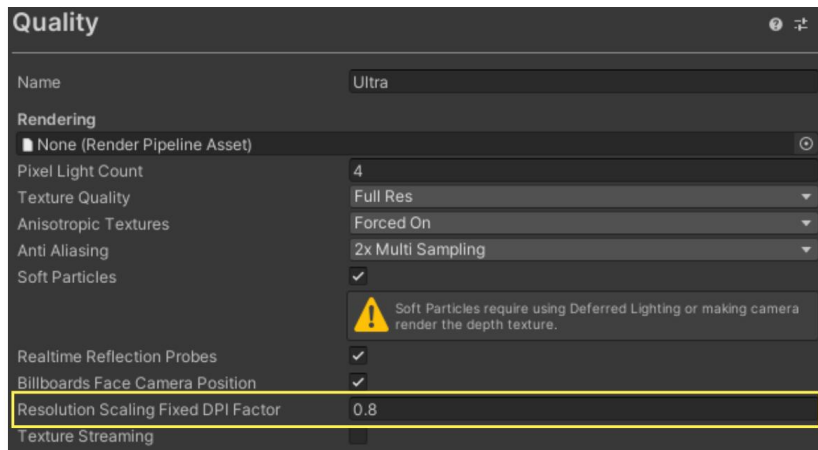
7.1.1 DPI Settings

If Resolution Scaling Mode, which is included in the resolution-related section of Player Settings for mobile platforms, is set to **Fixed DPI**, the specific **DPI (dots per inch)** The resolution can be reduced to target a specific DPI (dots per inch).



▲ Figure 7.1 Resolution Scaling Mode

The final resolution is determined by multiplying the Target DPI value by the Resolution Scaling DPI Scale Factor value in the Quality Settings.



▲ Figure 7.2 Resolution Scaling DPI Scale Factor

7.1.2 Resolution Scaling by Script

To dynamically change the drawing resolution from a script, call `Screen.SetResolution`.

The current resolution can be obtained at `Screen.width` or `Screen.height`, and DPI can be obtained at `Screen.dpi`.

▼ List 7.1 `Screen.SetResolution`

```
1: public void SetupResolution()
2: {
3:     var factor = 0.8f;
4:
5:     // Get current resolution with Screen.width, Screen.height
6:     var width = (int)(Screen.width * factor);
7:     var height = (int)(Screen.height * factor);
8:
9:     // Set Resolution
10:    Screen.SetResolution(width, height, true);
11: }
```


Resolution settings at `Screen.SetResolution` are reflected only on the actual device.

Note that changes are not reflected in the Editor.

7.2 Semi-transparency and overdraw

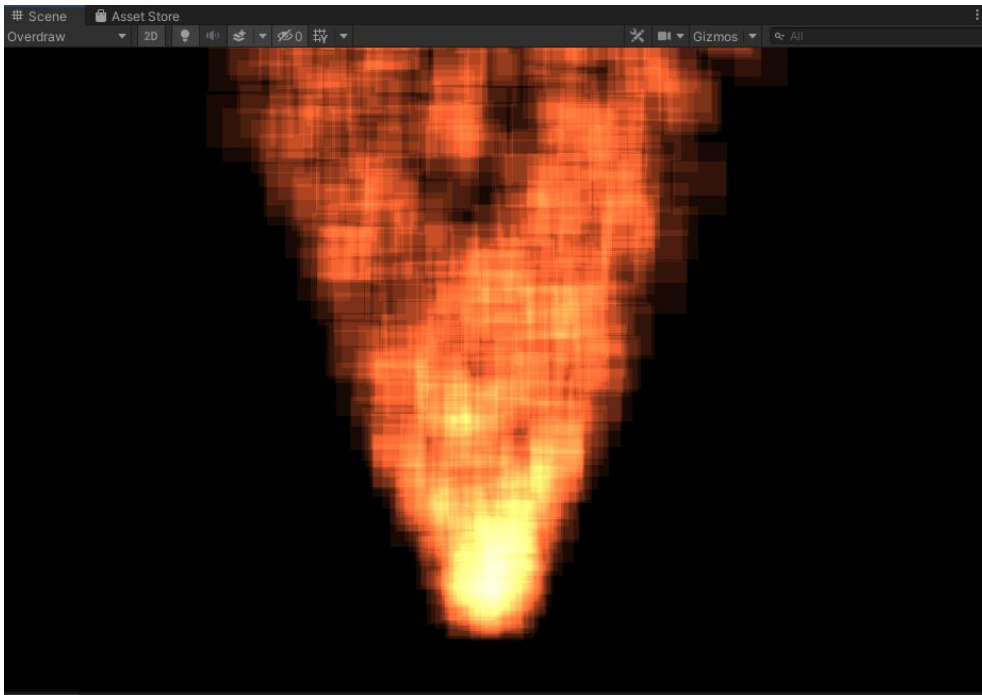
The use of translucent materials is controlled by the **overdraw** overdraw. Overdraw is the drawing of a fragment multiple times per pixel on the screen, and it affects performance in proportion to the load on the fragment shader.

Particularly when a large number of translucent particles are generated, such as in a particle system, a large amount of overdraw is often generated.

The following methods can be used to reduce the increased drawing load caused by overdraws.

- Reduce unnecessary drawing area
 - Reduce as much as possible the number of areas where textures are completely transparent, as they are also subject to rendering.
- Use lightweight shaders for objects that may cause overdraw.
- Avoid using semi-transparent materials as much as possible.
 - Use opaque materials to simulate the appearance of translucency **Dithering** is another technique to consider.

In the Editor of the Built-in Render Pipeline, set the Scene view mode to **Overdraw** in the Editor of the Built-in Render Pipeline, which is useful as a basis for adjusting overdraw.



▲ Figure 7.3 Overdraw mode

The Universal Render Pipeline supports the **Scene Debug View Modes** implemented in the Universal Render Pipeline since Unity 2021.2.

7.3 Reducing Draw Calls

Increasing the number of draw calls often affects the CPU load. Unity has several features to reduce the number of draw calls.

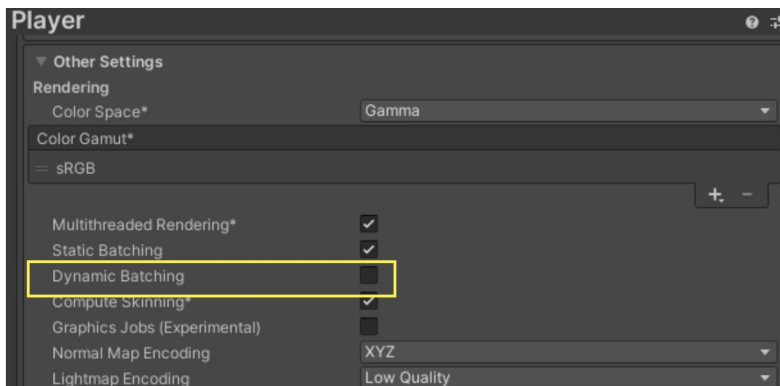
7.3.1 Dynamic batching

Dynamic batching is a feature for batching dynamic objects at runtime. This feature can be used to consolidate and reduce draw calls on dynamic objects that use the

same material.

To use it, go to Player Settings and select **Dynamic Batching** item in the Player Settings.

Also, in the Universal Render Pipeline, you can enable **Dynamic Batching** item in the Universal Render Pipeline Asset. However, the use of Dynamic Batching is deprecated in the Universal Render Pipeline.



▲ Figure 7.4 Dynamic Batching Settings

Because dynamic batching is a CPU-intensive process, many conditions must be met before it can be applied to an object. The main conditions are listed below.

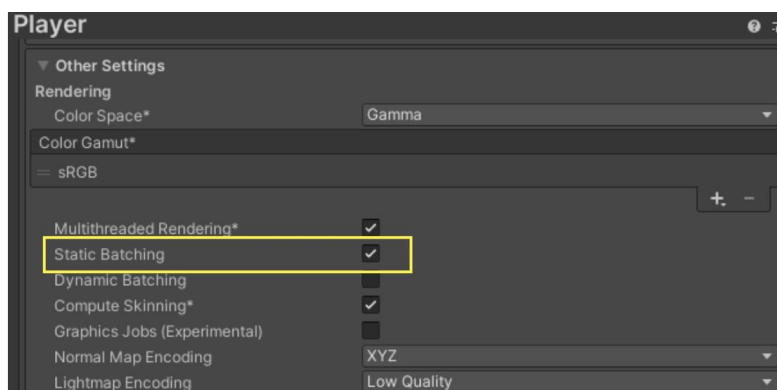
- Reference to the same material
- The object is being rendered with MeshRenderer or Particle System.
 - Other components such as SkinnedMeshRenderer are not subject to dynamic batching
- The number of mesh vertices is less than 300.
- No multipath is used
- Not affected by real-time shadows

Dynamic batching may not be recommended because of its impact on steady CPU load. See below. **SRP Batcher** described below can be used to achieve an effect similar to dynamic batching.

7.3.2 Static batching

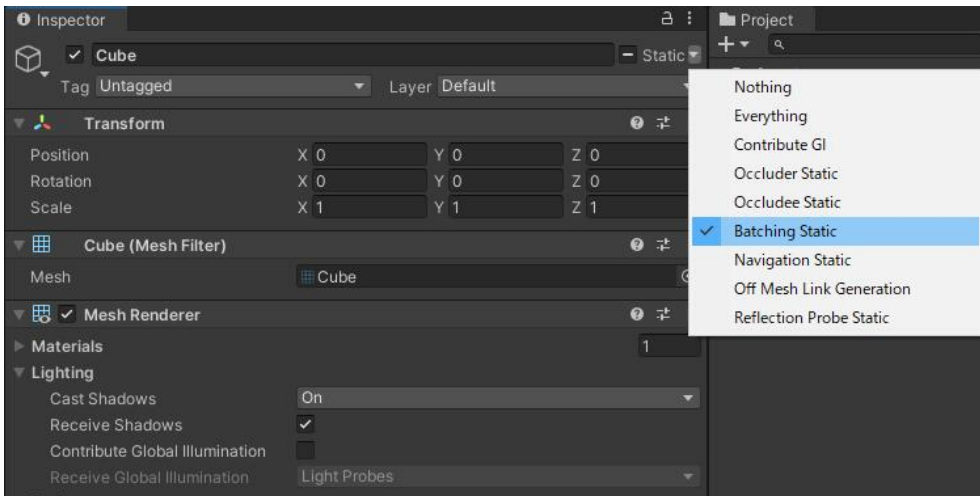
Static batching is a function for batching objects that do not move in the scene. This feature can be used to reduce draw calls on static objects using the same material.

Similarly to dynamic batching, from the Player Settings, click on the **Static Batching** from the Player Settings.



▲ Figure 7.5 Static Batching Settings

To make an object eligible for static batching, set the object's **static flag** flag of the object must be enabled. Specifically, the **Batching Static** sub-flag in the static flag must be enabled.



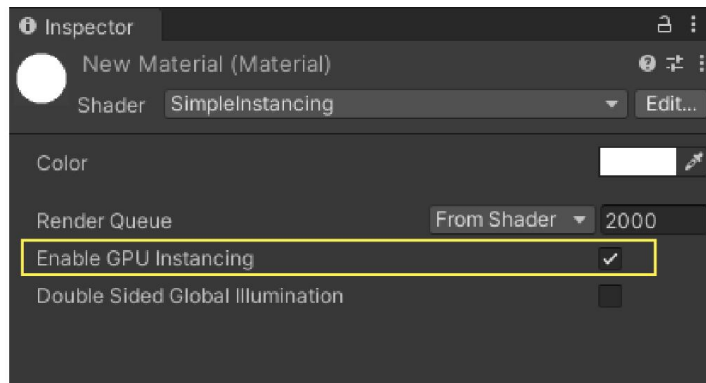
▲ Figure 7.6 Batching Static

Static batching differs from dynamic batching in that it does not involve vertex conversion processing at runtime, so it can be performed with a lower load. However, it should be noted that it consumes a lot of memory to store the mesh information combined by batch processing.

7.3.3 GPU Instancing

GPU instancing is a function for efficiently drawing objects of the same mesh and material. It is expected to reduce draw calls when drawing the same mesh multiple times, such as grass or trees.

To use GPU instancing, go to the material's Inspector and click on **Enable Instancing** from the material's Inspector.



▲ Figure 7.7 Enable Instancing

Creating shaders that can use GPU instancing requires some special handling. Below is an example shader code with a minimal implementation for using GPU instancing in a built-in render pipeline.

▼ List 7.2 Shaders that support GPU instancing

```
1: Shader "SimpleInstancing"
2: {
3:   Properties
4:   {
5:     _Color ("Color", Color) = (1, 1, 1, 1)
6:   }
7:
8:   CGINCLUDE
9:
10:  #include "UnityCG.cginc"
11:
12:  struct appdata
13:  {
14:    float4 vertex : POSITION;
15:    UNITY_VERTEX_INPUT_INSTANCE_ID
16:  };
17:
18:  struct v2f
19:  {
20:    float4 vertex : SV_POSITION;
21:    // Required only when accessing INSTANCED_PROP in fragment shaders
22:    UNITY_VERTEX_INPUT_INSTANCE_ID
23:  };
24:
25:  UNITY_INSTANCING_BUFFER_START(Props)
26:    UNITY_DEFINE_INSTANCED_PROP(float4, _Color)
27:  UNITY_INSTANCING_BUFFER_END(Props)
28:
29:  v2f vert(appdata v)
```

```

30:     {
31:         v2f o;
32:
33:         UNITY_SETUP_INSTANCE_ID(v);
34:
35:         // Required only when accessing INSTANCED_PROP in fragment shaders
36:         UNITY_TRANSFER_INSTANCE_ID(v, o);
37:
38:         o.vertex = UnityObjectToClipPos(v.vertex);
39:         return o;
40:     }
41:
42:     fixed4 frag(v2f i) : SV_Target
43:     {
44:         // Only required when accessing INSTANCED_PROP with fragment shaders
45:         UNITY_SETUP_INSTANCE_ID(i);
46:
47:         float4 color = UNITY_ACCESS_INSTANCED_PROP(Props, _Color);
48:         return color;
49:     }
50:
51:     ENDCG
52:
53:     SubShader
54:     {
55:         Tags { "RenderType"="Opaque" }
56:         LOD 100
57:
58:         Pass
59:         {
60:             CGPROGRAM
61:             #pragma vertex vert
62:             #pragma fragment frag
63:             #pragma multi_compile_instancing
64:             ENDCG
65:         }
66:     }
67: }

```

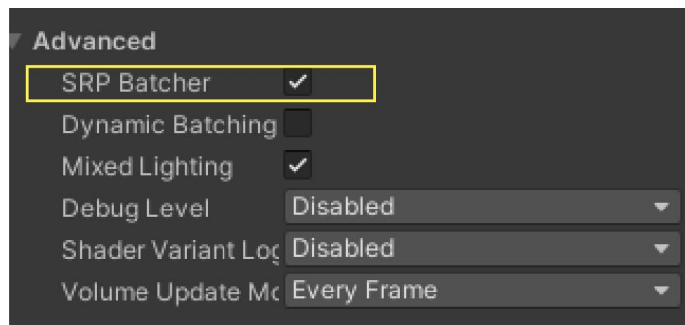
GPU instancing only works on objects that reference the same material, but you can set properties for each instance. You can set the target property as a property to be changed individually by enclosing it with `UNITY_INSTANCING_BUFFER_START(Props)` and `UNITY_INSTANCING_BUFFER_END(Props)`, as in the shader code above.

This property can then be set in C# to **MaterialPropertyBlock** API in C# to set properties such as individual colors. Just be careful not to use `MaterialPropertyBlock` for too many instances, as accessing the `MaterialPropertyBlock` may affect CPU performance.

7.3.4 SRP Batcher

SRP Batcher is a Scriptable Render Pipeline (SRP) that is used in the **Scriptable Render Pipeline** is a feature to reduce the CPU cost of rendering that is only available in the Scriptable Render Pipeline. This feature allows multiple shader set-pass calls that use the same shader variant to be processed together.

To use the SRP Batcher, you need to add the **Scriptable Render Pipeline Asset** from the Inspector of the **SRP Batcher** from the Inspector of the Scriptable Render Pipeline Asset.



▲ Figure 7.8 Enabling the SRP Batcher

You can also enable or disable the SRP Batcher at runtime with the following C# code

▼ List 7.3 Enabling SRP Batcher

```
1: GraphicsSettings.useScriptableRenderPipelineBatching = true;
```

The following two conditions must be met to make shaders compatible with SRP Batcher

1. Define built-in properties defined per object in a single CBUFFER 2. **called 1. UnityPerDraw**
2. Define properties per material in a single CBUFFER **called UnityPerMaterial**

For UnityPerDraw Universal Render Pipeline and other shaders basically support it by default, but you need to set up your own CBUFFER for UnityPerMaterial.

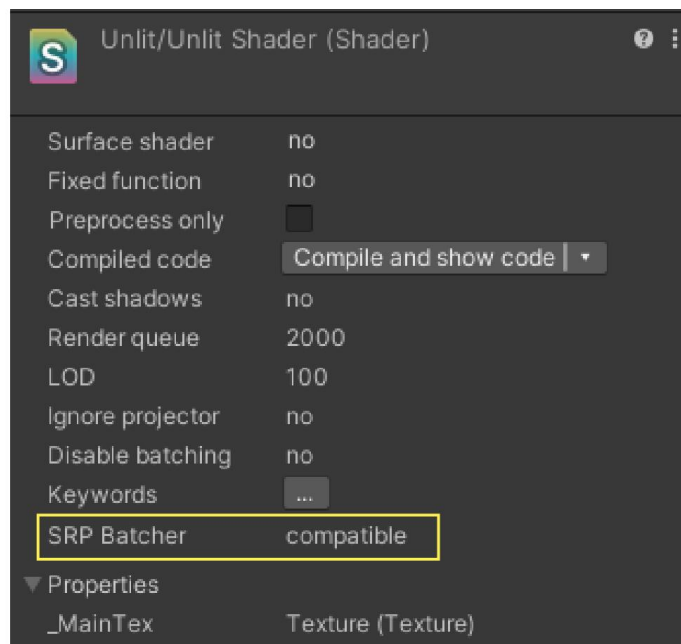
Surround the properties for each material with `CBUFFER_START(UnityPerMaterial 1)` and `CBUFFER_END` as shown below.

▼ List 7.4 UnityPerMaterial

```
1: Properties
2: {
3:     _Color1 ("Color 1", Color) = (1,1,1,1)
4:     _Color2 ("Color 2", Color) = (1,1,1,1)
5: }
6:
7: CBUFFER_START(UnityPerMaterial)
8:
9: float4 _Color1;
10: float4 _Color2;
11:
12: CBUFFER_END
```

With the above actions, you can create a shader that supports SRP Batcher, but you can also check if the shader in question supports SRP Batcher from Inspector.

In the Inspector of the shader, click on the **SRP Batcher** item in the shader's Inspector is **compatible** the shader is compatible with SRP Batcher, and **If it is "not compatible"** is not compatible, it means it is not supported.



▲ Figure 7.9 Shaders that are compatible with SRP Batcher

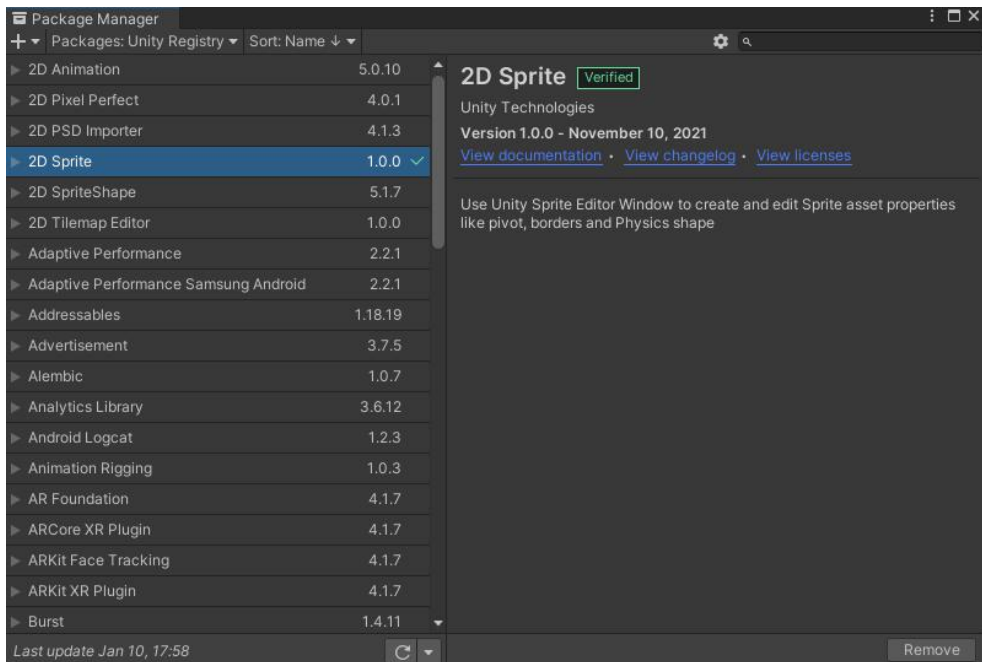
7.4 SpriteAtlas

2D games and UIs often use many sprites to build the screen. In such cases, a function to avoid generating a large number of draw calls is **SpriteAtlas** to avoid a large number of draw calls in such cases.

SpriteAtlas reduces draw calls by combining multiple sprites into a single texture.

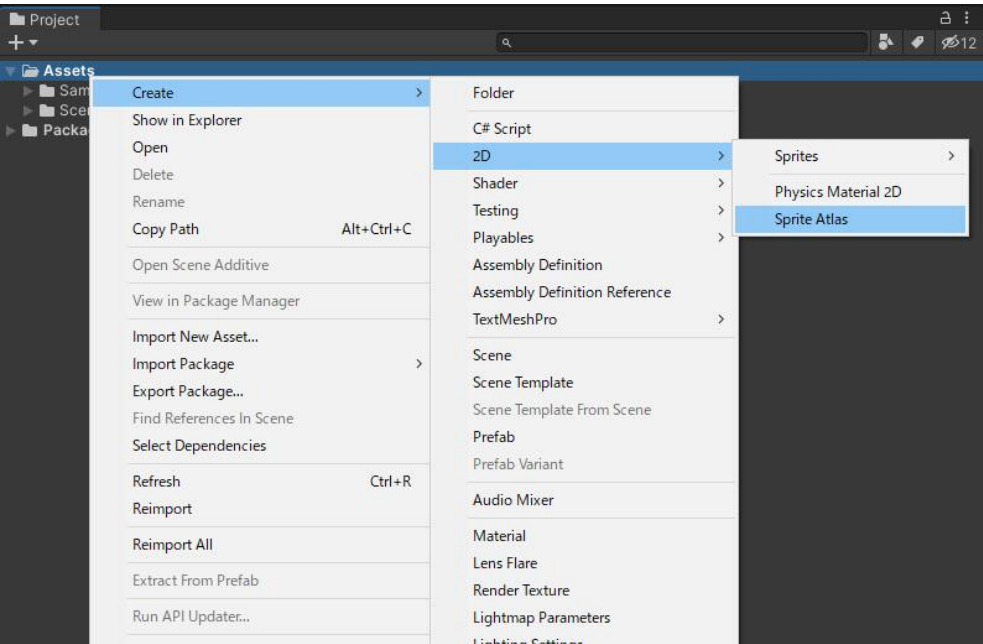
To create a SpriteAtlas, first go to the Package Manager and click on **2D Sprite** must first be installed in the project from the Package Manager.

7.4 SpriteAtlas



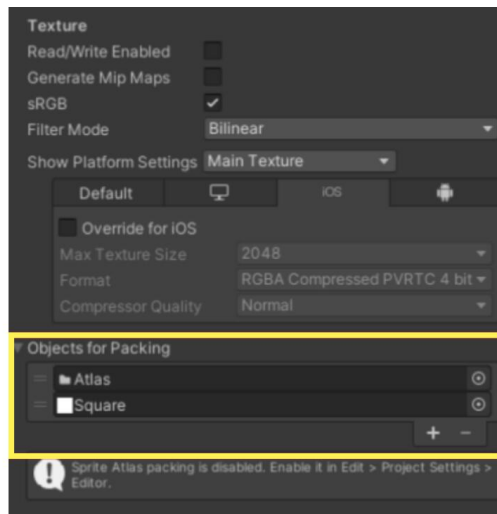
▲ Figure 7.10 2D Sprite

After installation, right click in the Project view and select "Create -> 2D -> Sprite Atlas" to create the SpriteAtlas asset.



▲ Figure 7.11 Creating SpriteAtlas

To specify the sprites that will be made into an atlas, go to the SpriteAtlas inspector and select **Objects for Packing** item of the SpriteAtlas inspector to specify the sprite or the folder that contains the sprite.



▲ Figure 7.12 Setting up Objects for Packing

With the above settings, the sprite will be atlased during build and playback in the Unity Editor, and the integrated SpriteAtlas texture will be referenced when drawing the target sprite.

Sprites can also be obtained directly from SpriteAtlas with the following code.

▼ List 7.5 Loading a Sprite from SpriteAtlas

```
1: [SerializeField]
2: private SpriteAtlas atlas;
3:
4: public Sprite LoadSprite(string spriteName)
5: {
6:     // Obtain a Sprite from SpriteAtlas with the Sprite name as an argument
7:     var sprite = atlas.GetSprite(spriteName);
8:     return sprite;
9: }
```

Loading a single Sprite in the SpriteAtlas consumes more memory than loading just one, since the texture of the entire atlas is loaded. Therefore, the SpriteAtlas should be used with care and divided appropriately.

This section is written targeting SpriteAtlas V1. SpriteAtlas V2 may have significant changes in operation, such as not being able to specify the folder of the sprite to be atlased.

7.5 Culling

In Unity, to omit in advance the processing of the parts that will not be displayed on the screen in the final version. **culling** process is used to eliminate the part of the image that will not ultimately be displayed on the screen in advance.

7.5.1 Visual Culling

Visual Culling is a process that omits objects outside of the camera's rendering area, the viewing cone, from the rendering. This prevents objects outside the camera's range from being calculated for rendering.

Visual cone culling is performed by default without any settings. For vertex shader-intensive objects, culling can be applied by dividing the mesh appropriately to reduce the cost of rendering.

7.5.2 Rear Culling

Rear culling is the process of omitting from rendering the backside of polygons that are (supposed to be) invisible to the camera. Most meshes are closed (only the front polygons are visible to the camera), so the backs of polygons do not need to be drawn.

In Unity, if you do not specify this in the shader, the back side of the polygon is subject to culling, but you can switch the culling setting by specifying it in the shader. The following is described in the SubShader.

▼ List 7.6 Culling setting

```
1: SubShader
2: {
3:     Tags { "RenderType"="Opaque" }
4:     LOD 100
5:
```

```
6:    Cull Back // Front, Off
7:
8:    Pass
9:    {
10:       CGPROGRAM
11:       #pragma vertex vert
12:       #pragma fragment frag
13:       ENDCG
14:    }
15: }
```

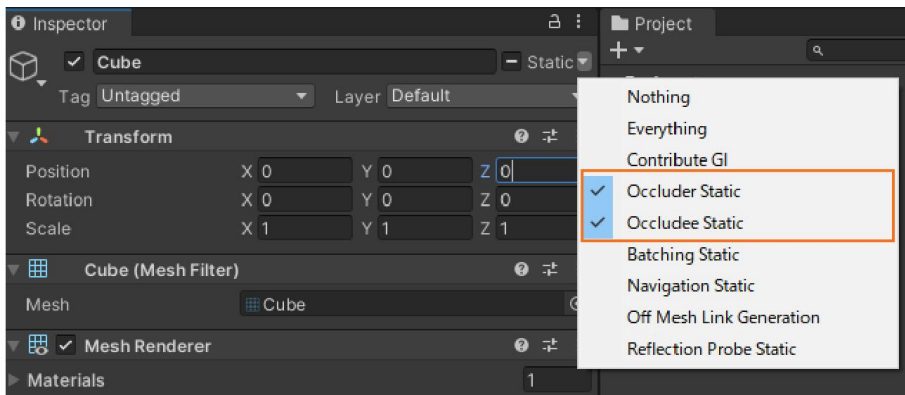
There are three settings: **Back**, **Front**, and **Off**. The effect of each setting is as follows.

- **Back** - Do not draw polygons on the side opposite to the viewer's point of view
- **Front** - Do not draw polygons in the same direction as the viewpoint
- **Off** - Disable back culling and draw all faces.

7.5.3 Occlusion culling

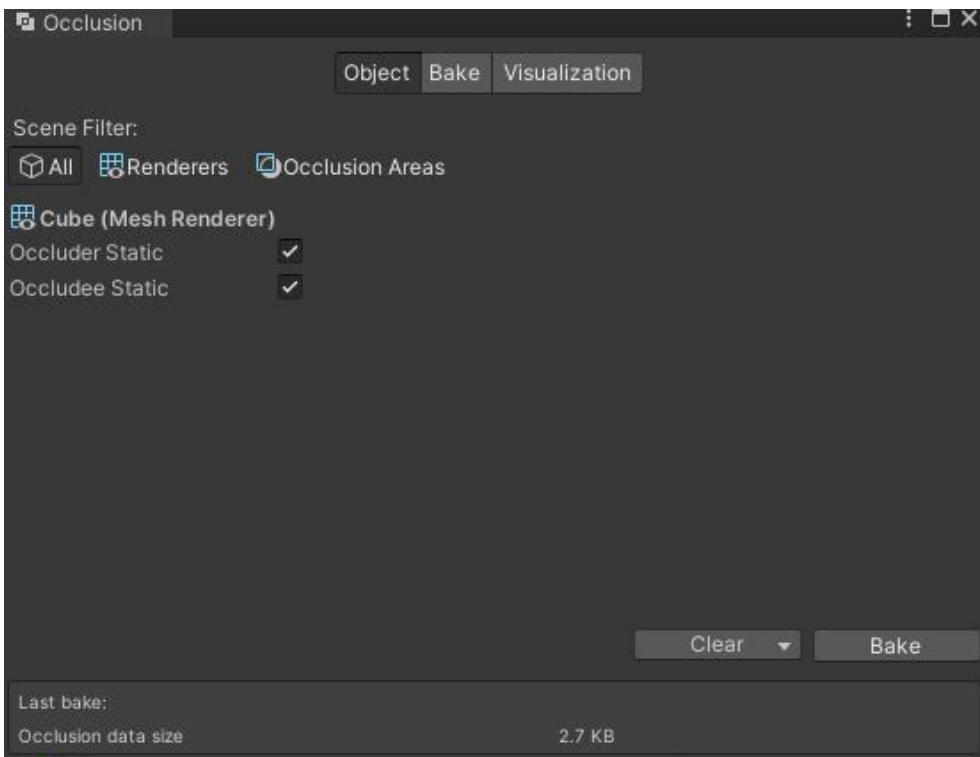
Occlusion culling is the process of omitting objects from the rendering that are not visible to the camera because they are occluded by objects. This function uses pre-baked occlusion data to determine if an object is occluded at run-time and removes the occluded object from the rendering.

To make an object eligible for occlusion culling, set the inspector's static flag to **Occluder Static** or **Occludee Static** from the inspector static flag. If **Occluder Static** is disabled and **Occludee Static** is enabled, the object will no longer be considered as the occluder, but **only as the occluded** object. In the opposite case, the object is no longer considered as a cloaked object and is processed as a **cloaked object only**.



▲ Figure 7.13 static flag for occlusion culling

To pre-bake for occlusion culling, the **Occlusion Culling window** is displayed to pre-bake for occlusion culling. In this window, you can change the static flags for each object, change the bake settings, etc., and press the **Bake button**. Bake can be performed by pressing the Bake button.



▲ Figure 7.14 Occlusion Culling Window

Occlusion culling reduces rendering cost, but at the same time, it puts more load on the CPU for the culling process, so it is necessary to balance each load and make appropriate settings.

Only the object rendering process is reduced by occlusion culling, while processes such as real-time shadow rendering remain unchanged.

7.6 Shaders

Shaders are very effective for graphics, but they often cause performance problems.

7.6.1 Reducing the precision of floating-point types

GPUs (especially on mobile platforms) compute faster with smaller data types than with larger ones. Therefore, floating-point types should be replaced with **float type (32bit)** to **half type (16bit)** is effective when it is possible to replace the floating-point type.

The float type should be used when precision is required, such as in depth calculations, but in color calculations, even if the precision is reduced, it is difficult to cause a large difference in the resulting appearance.

7.6.2 Performing Calculations with Vertex Shaders

The vertex shader is executed for the number of vertices in the mesh, and the fragment shader is executed for the number of pixels that will eventually be written. In general, vertex shaders are often executed less frequently than fragment shaders, so it is best to perform complex calculations in the vertex shader whenever possible.

The vertex shader calculation results are passed to the fragment shader via shader semantics, but it should be noted that the values passed are interpolated and may look different than if they were calculated in the fragment shader.

▼ List 7.7 Precomputation with vertex shaders

```
1: CGPROGRAM
2: #pragma vertex vert
3: #pragma fragment frag
4:
5: #include "UnityCG.cginc"
6:
7: struct appdata
8: {
9:     float4 vertex : POSITION;
10:    float2 uv : TEXCOORD0;
11: };
12:
13: struct v2f
14: {
15:     float2 uv : TEXCOORD0;
16:     float3 factor : TEXCOORD1;
17:     float4 vertex : SV_POSITION;
18: };
19:
20: sampler2D _MainTex;
21: float4 _MainTex_ST;
22:
```

```

23: v2f vert (appdata v)
24: {
25:     v2f o;
26:     o.vertex = UnityObjectToClipPos(v.vertex);
27:     o.uv = TRANSFORM_TEX(v.uv, _MainTex);
28:
29:     // Complex precomputations.
30:     o.factor = CalculateFactor();
31:
32:     return o;
33: }
34:
35: fixed4 frag (v2f i) : SV_Target
36: {
37:     fixed4 col = tex2D(_MainTex, i.uv);
38:
39:     // Values computed in the vertex shader are used in the fragment shader
40:     col *= i.factor;
41:
42:     return col;
43: }
44: ENDCG

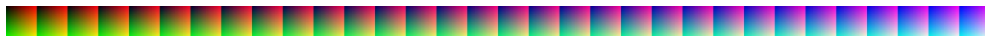
```

7.6.3 Prebuild information into textures

If the results of complex calculations in the shader are not affected by external values, then storing the pre-calculated results as elements in the texture is an effective way to do so.

This can be done by implementing a dedicated texture generation tool in Unity or as an extension to various DCC tools. If the alpha channel of a texture already in use is not being used, it is a good idea to write to it or prepare a dedicated texture.

For example, the **LUT (color correspondence table)** used for color grading, for example, will pre-correct the texture so that the coordinates of each pixel correspond to each color. By sampling the texture in the shader based on the original color, the result is almost the same as if the pre-correction had been applied to the original color.

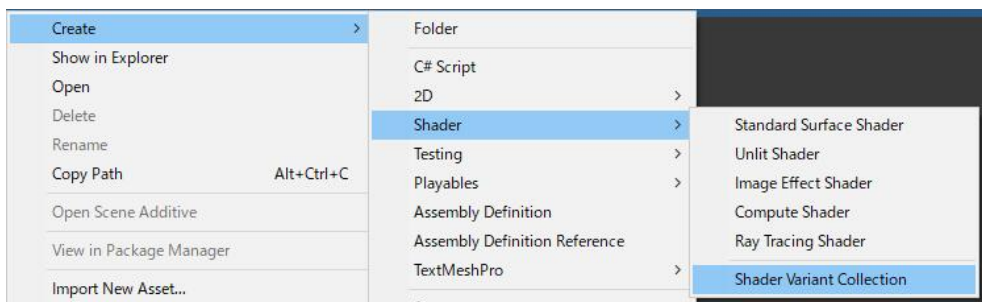


▲ Figure 7.15 LUT texture (1024x32) before color correction

7.6.4 ShaderVariantCollection

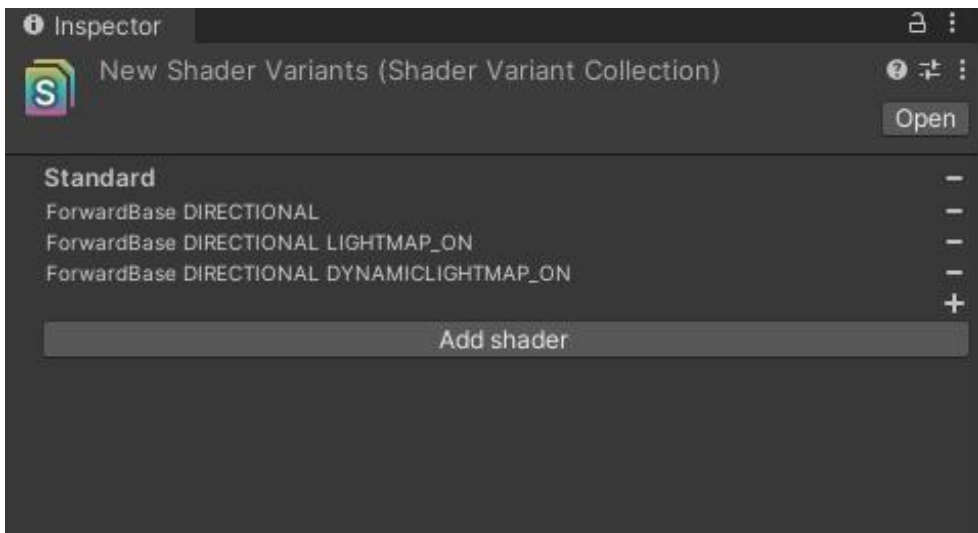
ShaderVariantCollection can be used to compile shaders before they are used to prevent spikes.

ShaderVariantCollection allows you to keep a list of shader variants used in your game as assets. It is created by selecting "Create -> Shader -> Shader Variant Collection" from the Project view.



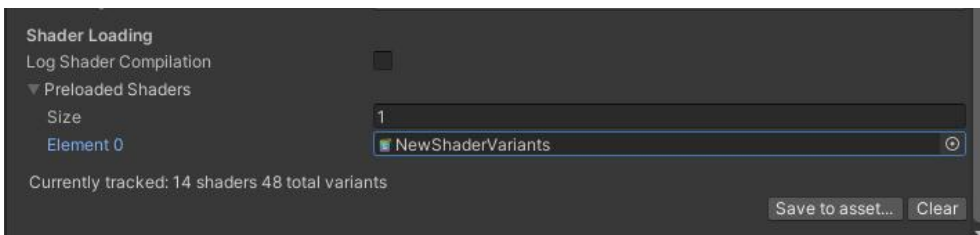
▲ Figure 7.16 Creating a ShaderVariantCollection

From the Inspector view of the created ShaderVariantCollection, press Add Shader to add the target shader, and then select which variants to add for the shader.



▲ Figure 7.17 ShaderVariantCollection Inspector

ShaderVariantCollection is added to the Graphics Settings **Shader preloading** in the Graphics Settings. **Preloaded Shaders** in the Shader preloading section of the Graphics Settings, to set the shader variants to be compiled at application startup.



▲ Figure 7.18 Preloaded Shaders

You can also set the shader variants from a script by calling **ShaderVariantCollection.WarmUp()** from a script to explicitly precompile the shader variants contained in the corresponding ShaderVariantCollection.

▼ List 7.8 ShaderVariantCollection.

```
1: public void PreloadShaderVariants(ShaderVariantCollection collection)
2: {
3:     // Explicitly precompile shader variants
4:     if (!collection.isWarmedUp)
5:     {
6:         collection.WarmUp();
7:     }
8: }
```

7.7 Lighting

Lighting is one of the most important aspects of a game's artistic expression, but it often has a significant impact on performance.

7.7.1 Real-time shadows

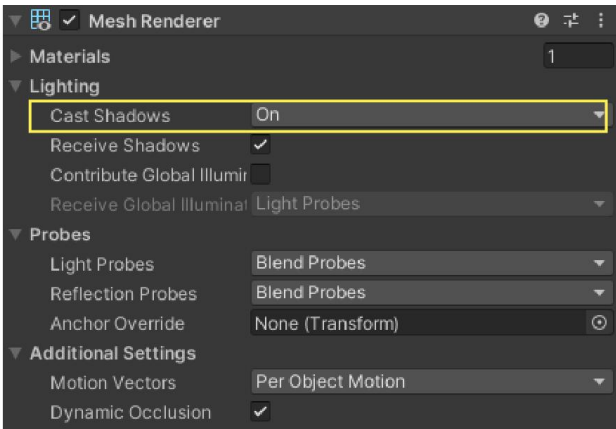
Generating real-time shadows consumes a large amount of draw call and fill rate. Therefore, careful consideration should be given to settings when using real-time shadows.

Draw Call Reduction

The following policies can be used to reduce draw calls for shadow generation.

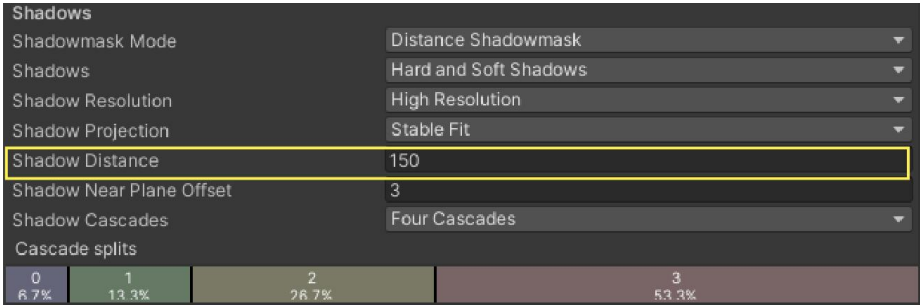
- Reduce the number of objects that drop shadows
- Consolidate draw calls by batching

There are several ways to reduce the number of objects dropping shadows, but a simple method is to use the MeshRenderer's **Cast Shadows** setting in MeshRenderer to off. This will remove the object from the shadow draw call. This setting is usually turned on in Unity and should be noted in projects that use shadows.



▲ Figure 7.19 Cast Shadows

It is also useful to reduce the maximum distance an object can be drawn in the shadow map. In the Quality Settings **Shadow Distance** in the Quality Settings to reduce the number of objects that cast shadows to the minimum necessary. Adjusting this setting will also reduce the resolution of the shadows, since shadows will be drawn at the minimum range for the resolution of the shadow map.



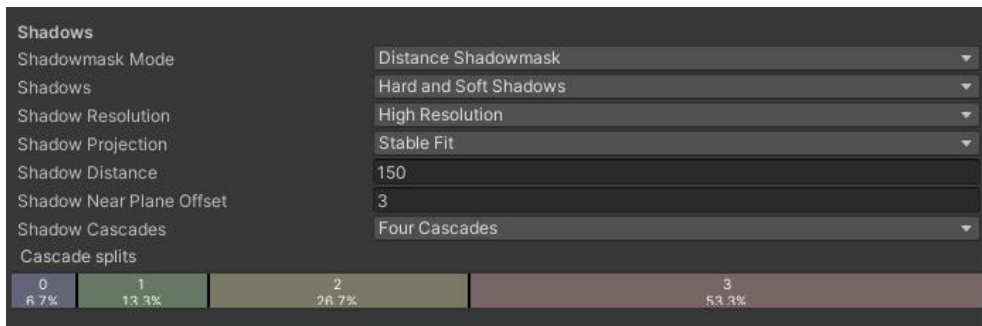
▲ Figure 7.20 Shadow Distance

As with normal rendering, shadow rendering can be subject to batching to reduce draw calls. See "7.3 Reducing Draw Calls" for more information on batching techniques.

Fill Rate Savings

The fill rate of a shadow depends on both the rendering of the shadow map and the rendering of the object affected by the shadow.

The respective fill rates can be saved by adjusting several settings in the Shadows section of the Quality Settings.

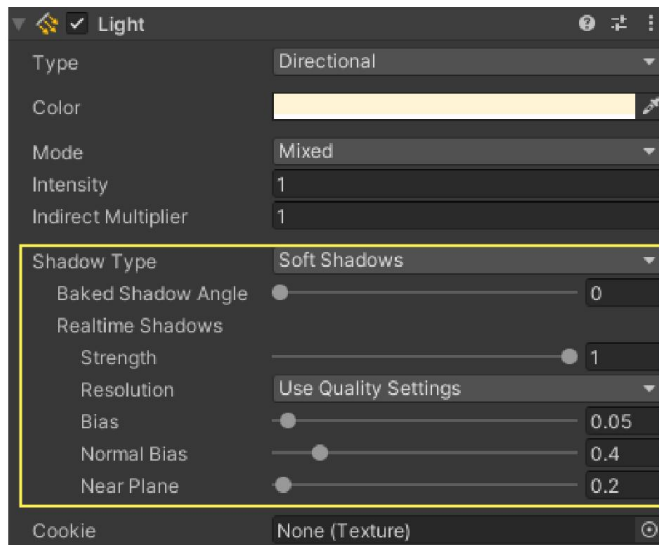


▲ Figure 7.21 Quality Settings -> Shadows

The **Shadows** section allows you to change the format of the shadows, and the **Hard Shadows** will produce a clear shadow border, but with a relatively low load, while **Soft Shadows** is more expensive, but it can produce blurred shadow borders.

Shadow Resolution and **Shadow Cascades** items affect the resolution of the shadow map, with larger settings increasing the resolution of the shadow map and consuming more fill rate. However, since these settings have a great deal to do with the quality of the shadows, they should be carefully adjusted to strike a balance between performance and quality.

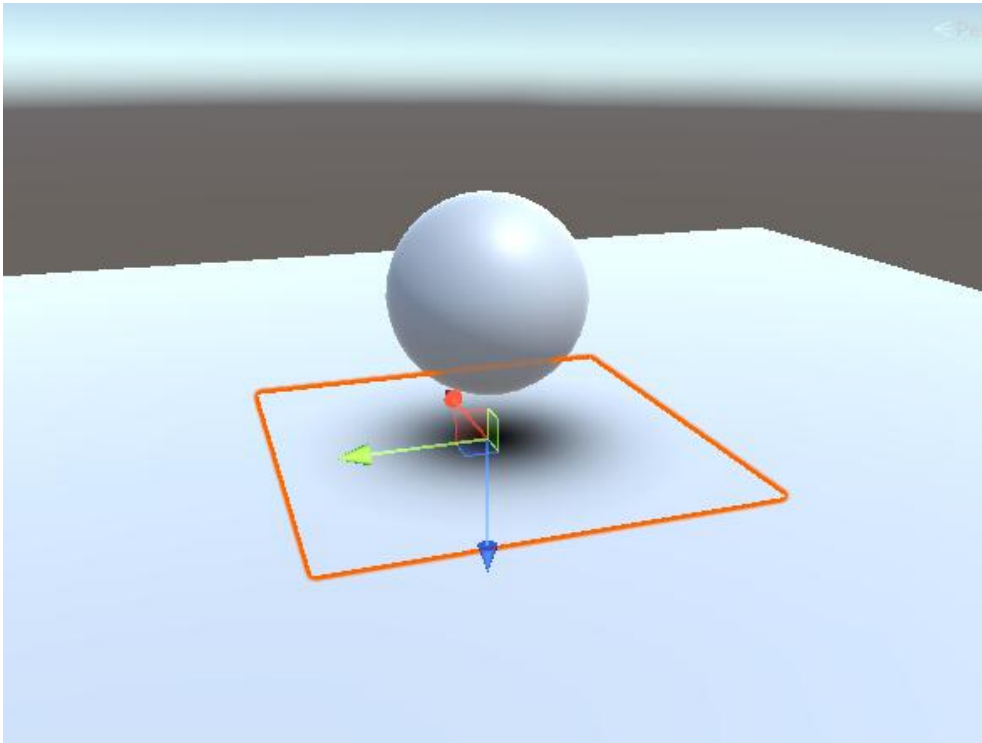
Some settings can be adjusted using the **Light** component's Inspector, so it is possible to change the settings for individual lights.



▲ Figure 7.22 Shadow settings for the Light component

Pseudo Shadow

Depending on the game genre or art style, it may be effective to use plane polygons or other materials to simulate the shadows of objects. Although this method has strong usage restrictions and is not highly flexible, it is far lighter than the usual real-time shadow rendering method.

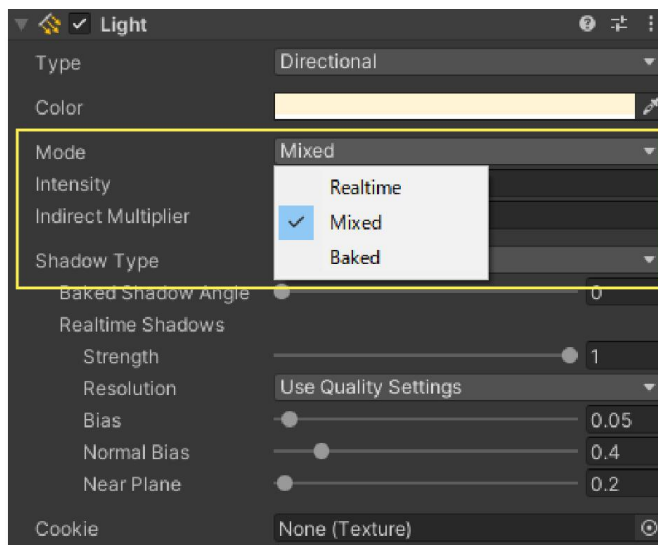


▲ Figure 7.23 Pseudo-shadow using plate polygons

7.7.2 Light Mapping

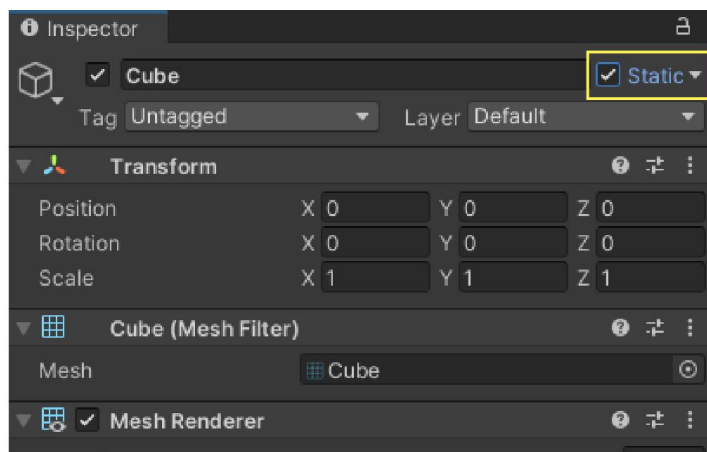
By baking lighting effects and shadows into textures in advance, quality lighting expressions can be achieved with considerably lower load than real-time generation.

To bake a lightmap, first add a Light component placed in the scene **Mode** item of the Light component placed in the scene to **Mixed** or **Baked** Mixed or Baked.



▲ Figure 7.24 Mode setting for Light

Also, activate the static flag of the object to be baked.

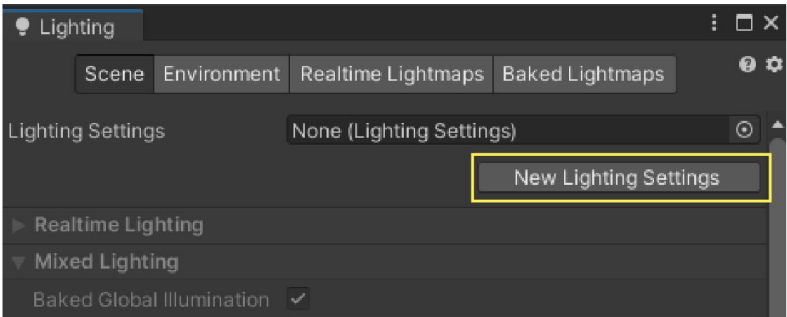


▲ Figure 7.25 Enable static

In this state, select "Window -> Rendering -> Lighting" from the menu to display the Lighting view.

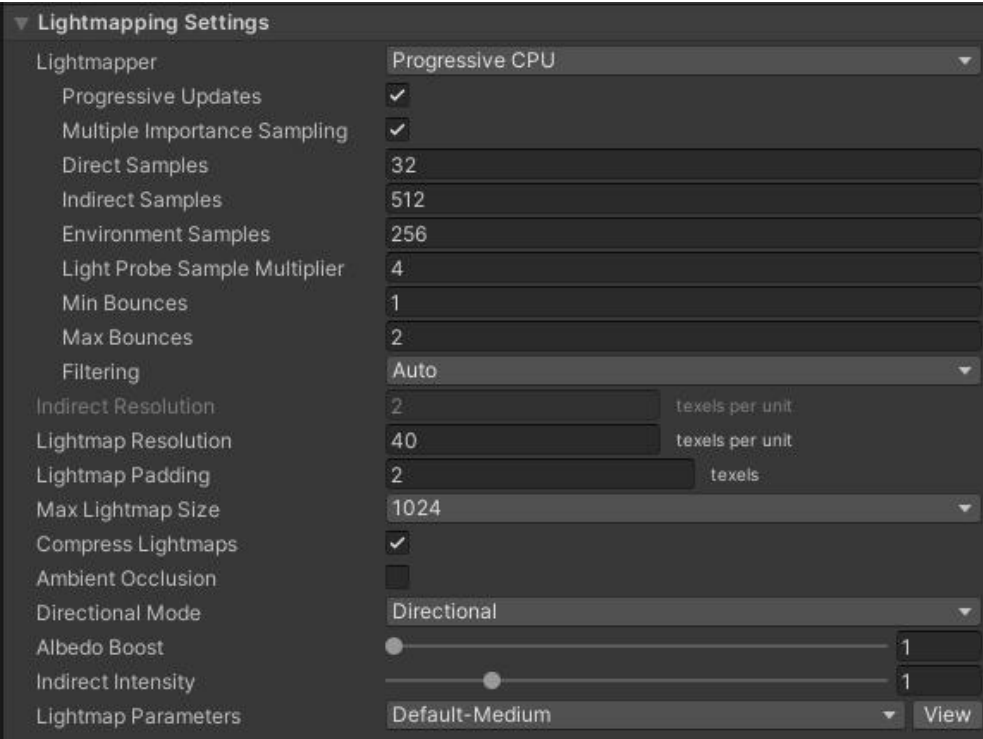
The default setting is **Lighting Settings** asset is not specified, we need to change

the settings by clicking on **New Lighting Settings** button to create a new one.



▲ Figure 7.26 New Lighting Settings

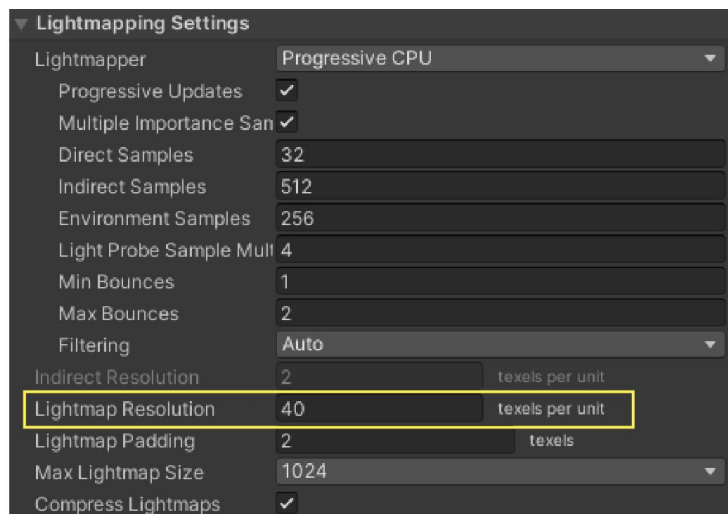
The main settings for lightmaps are **Lightmapping Settings** tab.



▲ Figure 7.27 Lightmapping Settings

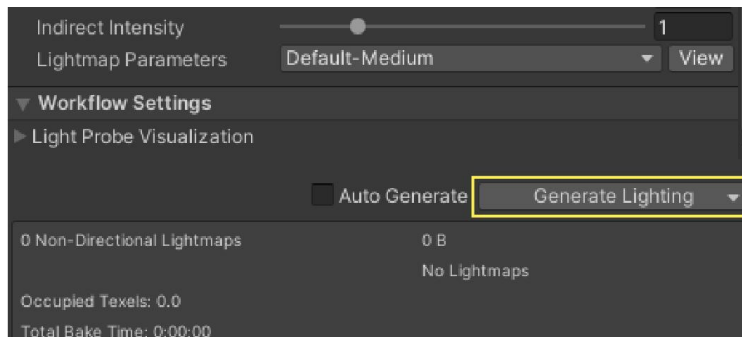
There are many settings that can be adjusted to change the speed and quality of lightmap baking. Therefore, these settings should be adjusted appropriately for the desired speed and quality.

Of these settings, the one that has the greatest impact on performance is **Lightmap Resolution**. This setting has the largest impact on performance. This setting determines how many lightmap texels are allocated per unit in Unity, and since the final lightmap size varies depending on this value, it has a significant impact on storage and memory capacity, texture access speed, and other factors.

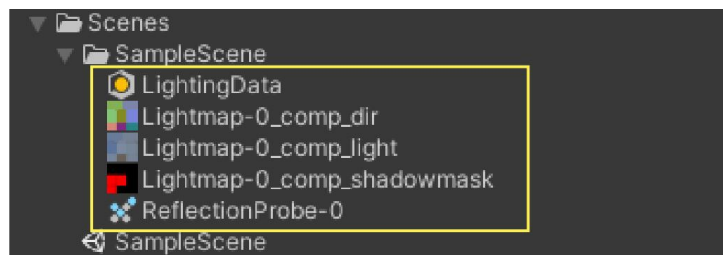


▲ Figure 7.28 Lightmap Resolution

Finally, at the bottom of the Inspector view, the **Generate Lighting** button at the bottom of the Inspector view to bake the lightmap. Once baking is complete, you will see the baked lightmap stored in a folder with the same name as the scene.



▲ Figure 7.29 Generate Lighting

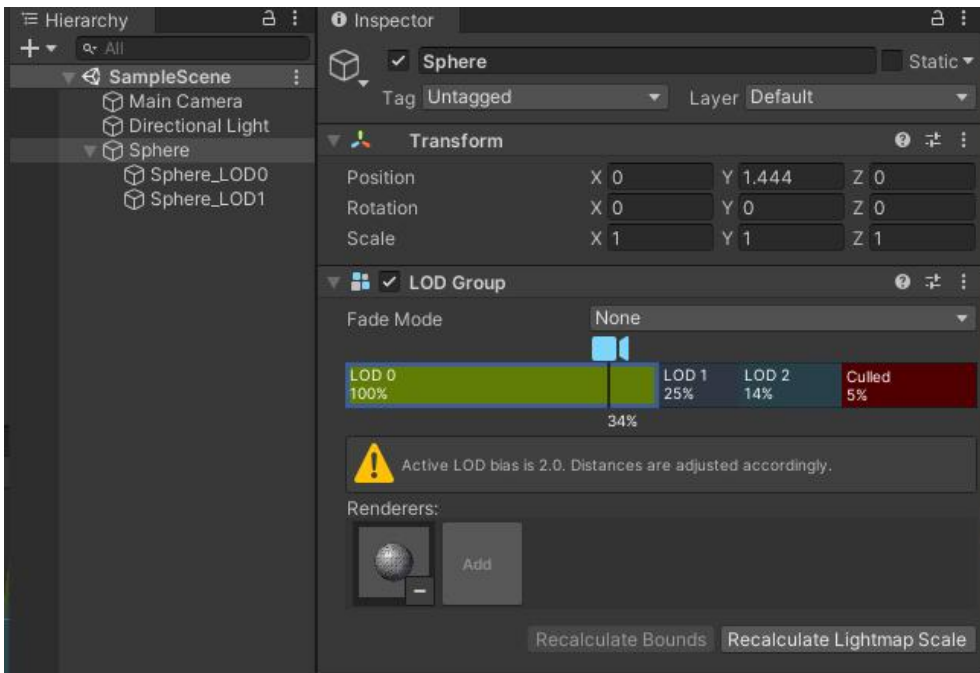


▲ Figure 7.30 Baked lightmap

7.8 Level of Detail

It is inefficient to render objects that are far away from the camera in high-polygon, high-definition. **Level of Detail (LOD)** method can be used to reduce the level of detail of an object depending on its distance from the camera.

In Unity, objects are assigned to a **LOD Group** component to the object.



▲ Figure 7.31 LOD Group

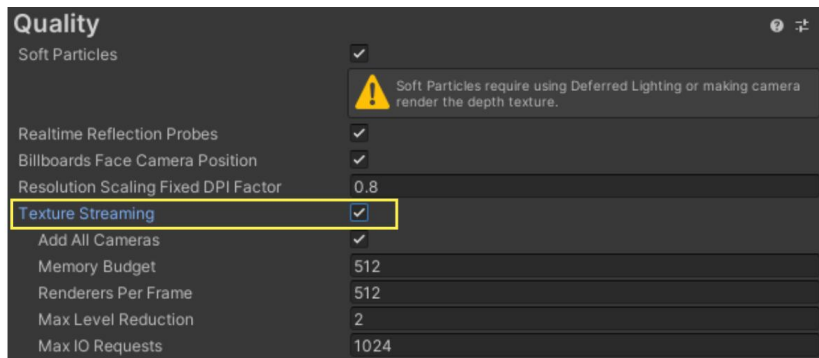
By placing a renderer with a mesh of each LOD level on a child of a GameObject to which a LOD Group is attached and setting each LOD level in the LOD Group, the LOD level can be switched according to the camera. It is also possible to set which LOD level is assigned to each LOD Group in relation to the camera's distance.

Using LOD generally reduces the drawing load, but one should be aware of memory and storage pressures since all meshes for each LOD level are loaded.

7.9 Texture Streaming

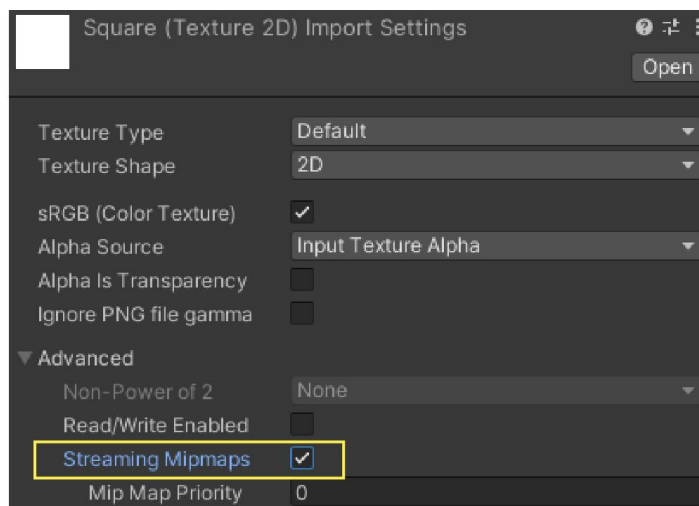
Unity's **Texture Streaming** can be used to reduce the memory footprint and load time required for textures. Texture streaming is a feature that saves GPU memory by loading mipmaps based on the camera position in the scene.

To enable this feature, go to the Quality Settings **Texture Streaming** in the Quality Settings.



▲ Figure 7.32 Texture Streaming

In addition, the texture import settings must be changed to allow streaming of texture mipmaps. Open the texture inspector and select **Advanced Streaming Mipmaps** in the Advanced settings.



▲ Figure 7.33 Streaming Mipmaps

These settings enable streaming mipmaps for the specified texture. Also, in the Quality Settings **Memory Budget** under Quality Settings to limit the total memory usage of the loaded textures. The texture streaming system will load the mipmaps without exceeding the amount of memory set here.



PERFORMANCE TUNING BIBLE

CHAPTER

08

第8章

Tuning Practice

— UI —

CyberAgent Smartphone Games & Entertainment

Chapter 8

Tuning Practice - UI

Tuning Practices for uGUI, the Unity standard UI system, and TextMeshPro, the mechanism for drawing text to the screen.

8.1 Canvas partitioning

In uGUI, when there is a change in an element in `Canvas`, a process (rebuild) runs to rebuild the entire `Canvas` UI mesh. A change is any change, such as active switching, moving or resizing, from a major change in appearance to a minor change that is not apparent at first glance. The cost of the rebuild process is high, so if it is performed too many times or if the number of UIs in `Canvas` is large, performance will be adversely affected.

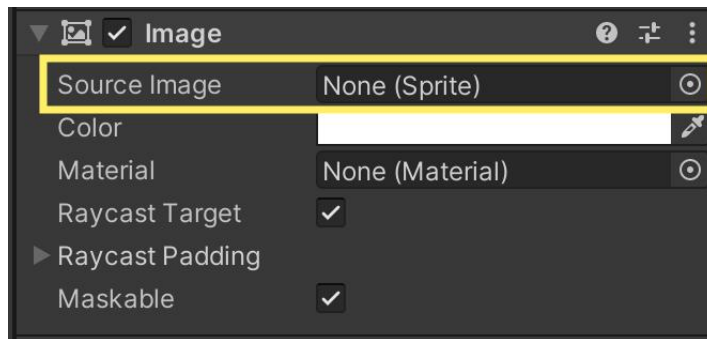
In contrast, the cost of rebuilds can be reduced by dividing `Canvas` by some degree of UI cohesion. For example, if you have UIs that animate and UIs that do not animate, you can minimize the animation rebuilds by placing them under a separate `Canvas`.

However, you need to think carefully about how to split them, as splitting `Canvas` will not work for drawing batches.

Splitting `Canvas` is also valid when `Canvas` is nested under `Canvas`. If the elements contained in the child `Canvas` change, a rebuild of the child `Canvas` will only run, not the parent `Canvas`. However, upon closer inspection, it seems that the situation is different when the UI in the child `Canvas` is switched to the active state by `SetActive`. In this case, if a large number of UIs are placed in the parent `Canvas`, there seems to be a phenomenon that causes a high load. I do not know the details of why this behavior occurs, but it seems that care should be taken when switching the active state of the UI in the nested `Canvas`.

8.2 UnityWhite

When developing UIs, it is often the case that we want to display a simple rectangle-shaped object. This is where UnityWhite comes in handy. UnityWhite is a Unity built-in texture that is used when the `Image` or `RawImage` component does not specify the image to be used (Figure 8.1). You can see how UnityWhite is used in the Frame Debugger (Figure 8.2). This mechanism can be used to draw a white rectangle, so a simple rectangle type display can be achieved by combining this with a multiplying color.



▲ Figure 8.1 Using UnityWhite



▲ Figure 8.2 UnityWhite in use.

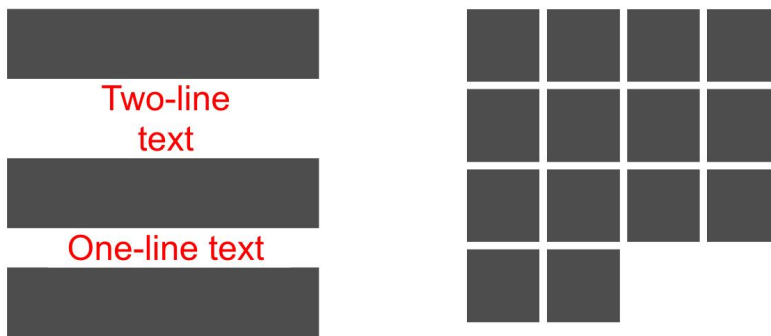
However, since UnityWhite is a different texture than the SpriteAtlas provided in the project, draw batches are interrupted. This increases draw calls and reduces drawing efficiency.

Therefore, you should add a small (e.g., 4 x 4 pixel) white square image to SpriteAtlas and use that Sprite to draw a simple rectangle. This will allow the batch to work,

since the same SpriteAtlas will be used for the same material.

8.3 Layout component

uGUI provides a Layout component that allows you to neatly align objects. For example, `VerticalLayoutGroup` is used for vertical alignment, and `GridLayoutGroup` is used for alignment on the grid (Figure 8.3).



▲ Figure 8.3 Example using `VerticalLayoutGroup` on the left and `GridLayoutGroup` on the right

When using the Layout component, Layout rebuilds occur when the target object is created or when certain properties are edited. Layout rebuilds, like mesh rebuilds, are costly processes.

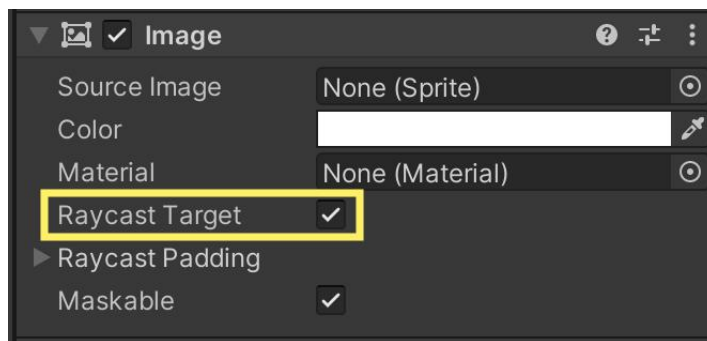
To avoid performance degradation due to Layout rebuilds, it is effective to avoid using Layout components as much as possible.

For example, if you do not need dynamic placement, such as text that changes placement based on its content, you do not need to use the Layout component. If you really need dynamic placement, or if it is used a lot on the screen, it may be better to control it with your own scripts. Also, if the requirement is to be placed in a specific position relative to the parent even if the parent changes size, this can be accomplished by adjusting the `RectTransform` anchors. If you use a Layout component when creating a prefab because it is convenient for placement, be sure

to delete it and save it.

8.4 Raycast Target

`Graphic`, the base class for `Image` and `RawImage`, has a property `Raycast Target` (Figure 8.4). When this property is enabled, its `Graphic` becomes the target for clicks and touches. When the screen is clicked or touched, objects with this property enabled will be the target of processing, so disabling this property as much as possible will improve performance.



▲ Figure 8.4 Raycast Target property

This property is enabled by default, but actually many `Graphic` do not require this property to be enabled. On the other hand, Unity has a feature called preset^{*1} that allows you to change the default value in your project. Specifically, you can create presets for the `Image` and `RawImage` components, respectively, and register them as default presets from the Preset Manager in Project Settings. You may also use this feature to disable the `Raycast Target` property by default.

8.5 Masks

To represent masks in uGUI, use either the `Mask` component or the `RectMask2d` component.

Since `Mask` uses stencils to realize masks, the drawing cost increases with each

^{*1} <https://docs.unity3d.com/ja/current/Manual/Presets.html>

additional component. On the other hand, `RectMask2d` uses shader parameters to realize masks, so the increase in drawing cost is suppressed. However, `Mask` can be hollowed out in any shape, while `RectMask2d` can only be hollowed out as a rectangle.

It is a common belief that `RectMask2d` should be selected if available, but recent Unity users should also be careful about using `RectMask2d`.

Specifically, when `RectMask2d` is enabled, the CPU load for culling every frame is proportional to as its masking target increases. This phenomenon, which generates load every frame even when the UI is not moving anything, seems to be a side effect of a fix to issue ^{*2} that was included in Unity 2019.3, according to comments in the uGUI's internal implementation.

Therefore, it is useful to take measures to avoid using `RectMask2d` as much as possible, to use `enabled` as `false` when it is not needed even if is used, and to keep masked targets to the minimum necessary.

8.6 TextMeshPro

The common way to set text in `TextMeshPro` is to assign text to the `text` property, but there is another method, `SetText`.

There are many overloads to `SetText`, for example, that take a string and a value of type `float` as arguments. If you use this method like List 8.1, you can print the value of the second argument. However, assume that `label` is a variable of type `TMP_Text`(or inherited from it) and `number` is of type `float`.

▼ List 8.1 Example of using `SetText`

```
1: label.SetText("{0}", number);
```

The advantage of this method is that it reduces the cost of generating strings.

▼ List 8.2 Example of not using `SetText`

```
1: label.text = number.ToString();
```

^{*2} <https://issuetracker.unity3d.com/issues/rectmask2d-diffrently-masks-image-in-the-play-mode-when-animating-rect-transform-pivot-property>

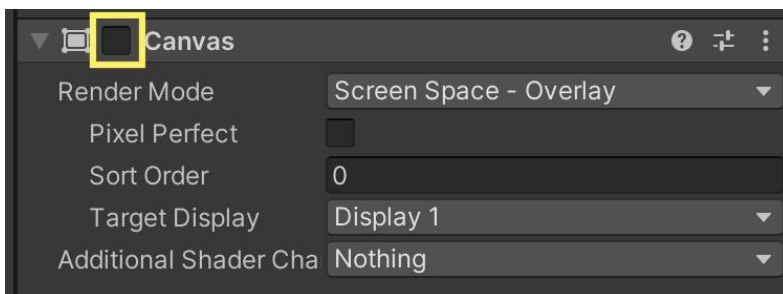
List 8.2 In the method using the `text` property, as in the following example, `ToString()` of type `float` is executed, so the string generation cost is incurred each time this process is executed. In contrast, the method using `SetText` is designed to generate as few strings as possible, which is a performance advantage when the text to be displayed changes frequently, as is the case with .

This feature of `TextMeshPro` is also very powerful when combined with `ZString`^{*3}. `ZString` is a library that reduces memory allocation in string generation. `ZString` provides many extension methods for the `TMP_Text` type, and by using those methods, flexible text display can be achieved while reducing the cost of string generation.

8.7 UI Display Switching

uGUI components are characterized by the high cost of active switching of objects by `SetActive`. This is due to the fact that `OnEnable` sets the Dirty flag for various rebuilds and performs initialization related to masks. Therefore, it is important to consider alternatives to the `SetActive` method for switching the display of the UI.

The first method is to change `enabled` of `Canvas` to `false` (Figure 8.5). This will prevent all objects under `Canvas` from being rendered. Therefore, this method has the disadvantage that it can only be used if you want to hide all the objects under `Canvas`.



▲ Figure 8.5 Canvas Disabling

Another method is to use `CanvasGroup`. `CanvasGroup` has a function that allows

^{*3} <https://github.com/Cysharp/ZString>

you to adjust the transparency of all objects under it at once. If you use this function and set the transparency to 0, you can hide all the objects under its `CanvasGroup`(Figure 8.6).



▲ Figure 8.6 `CanvasGroup` Set the transparency of the to 0.

While these methods are expected to avoid the load caused by `SetActive`, you may need to be careful because `GameObject` will remain in the active state. For example, if `Update` methods are defined, be aware that they will continue to run even in the hidden state, which may lead to an unexpected increase in load.

For reference, we measured the processing time for 1280 `GameObject` with `Image` components attached, when switching between visible and hidden states using each method (Table 8.1). The processing time was measured using the Unity editor, and Deep Profile was not used. The processing time of the method is the sum of the execution time of the actual switching^{*4} and the execution time of `UIEvents.WillRenderCanvases` in the frame. The execution time of `UIEvents.WillRenderCanvases` is added together because the UI rebuild is performed in.

▼ Table 8.1 Processing time for display switching

Method	Processing time (display)	Processing time (hidden)
<code>SetActive</code>	323.79ms	209.93ms
<code>Canvas Of enabled</code>	61.25ms	61.23ms
<code>CanvasGroup Of alpha</code>	3.64ms	3.40ms

Table 8.1 From the results of , we found that the method using `CanvasGroup` has by far the shortest processing time in the situation we tried this time.

^{*4} For example, if `SetActive`, the `SetActive` method call is enclosed in `Profiler.BeginSample` and `Profiler.EndSample` to measure the time.



PERFORMANCE TUNING BIBLE

CHAPTER

09

第9章

**Tuning Practice
— Script (Unity) —**

CyberAgent Smartphone Games & Entertainment

Chapter 9

Tuning Practice - Script (Unity)

Casual use of the features provided by Unity can lead to unexpected pitfalls. This chapter introduces performance tuning techniques related to Unity's internal implementation with actual examples.

9.1 Empty Unity event functions

When Unity-provided event functions such as *Awake*, *Start*, and *Update* are defined, they are cached in an internal Unity list at runtime and executed by iteration of the list.

Even if nothing is done in the function, it will be cached simply because it is defined. Leaving unneeded event functions in place will bloat the list and increase the cost of iteration.

For example, as shown in the sample code below, *Start* and *Update* are defined from the beginning in a newly generated script on Unity. If you do not need these functions, be sure to delete them.

▼ List 9.1 Newly generated script in Unity

```
1: public class NewBehaviourScript : MonoBehaviour
2: {
3:     // Start is called before the first frame update
4:     void Start()
5:     {
6:
7:     }
8:
9:     // Update is called once per frame
10:    void Update()
11:    {
12:
13:    }
14: }
```

9.2 Accessing tags and names

Classes inheriting from `UnityEngine.Object` provide the `tag` and `name` properties. These properties are useful for object identification, but in fact `GC.Alloc`.

I have quoted their respective implementations from `UnityCsReference`. You can see that both call processes implemented in native code.

Unity implements scripts in C#, but Unity itself is implemented in C++. Since C# memory space and C++ memory space cannot be shared, memory is allocated to pass string information from the C++ side to the C# side. This is done each time it is called, so if you want to access it multiple times, you should cache it.

For more information on how Unity works and memory between C# and C++, see "Unity Runtime".

▼ List 9.2 `UnityCsReference GameObject.bindings.cs` taken from ^{*1}

```
1: public extern string tag
2: {
3:     [FreeFunction("GameObjectBindings::GetTag", HasExplicitThis = true)]
4:     get;
5:     [FreeFunction("GameObjectBindings::SetTag", HasExplicitThis = true)]
6:     set;
7: }
```

▼ List 9.3 `UnityCsReference UnityEngineObject.bindings.cs` taken from ^{*2}

```
1: public string name
2: {
3:     get { return GetName(this); }
4:     set { SetName(this, value); }
5: }
6:
7: [FreeFunction("UnityEngineObjectBindings::GetName")]
8: extern static string GetName([NotNull("NullExceptionObject")] Object obj);
```

9.3 Retrieving Components

`GetComponent()`, which retrieves other components attached to the same `GameObj`

^{*1} ??

^{*2} ??

Chapter 9 Tuning Practice - Script (Unity)

ect, is another one that requires attention.

As well as the fact that it calls a process implemented in native code, similar to the tag and name properties in the previous section, we must also be careful about the cost of "searching" for components of the specified type.

In the sample code below, you will have the cost of searching for Rigidbody components every frame. If you access the site frequently, you should use a pre-cached version of the site.

▼ List 9.4 Code to GetComponent() every frame

```
1: void Update()
2: {
3:     Rigidbody rb = GetComponent<Rigidbody>();
4:     rb.AddForce(Vector3.up * 10f);
5: }
```

9.4 Accessing transform

Transform components are frequently accessed components such as position, rotation, scale (expansion and contraction), and parent-child relationship changes. As shown in the sample code below, you will often need to update multiple values.

▼ List 9.5 Example of accessing transform

```
1: void SetTransform(Vector3 position, Quaternion rotation, Vector3 scale)
2: {
3:     transform.position = position;
4:     transform.rotation = rotation;
5:     transform.localScale = scale;
6: }
```

When transform is retrieved, the process `GetTransform()` is called inside Unity. It is optimized and faster than `GetComponent()` in the previous section. However, it is slower than the cached case, so this should also be cached and accessed as shown in the sample code below. For position and rotation, you can also use `SetPositionAndRotation()` to reduce the number of function calls.

▼ List 9.6 Example of caching transform

```
1: void SetTransform(Vector3 position, Quaternion rotation, Vector3 scale)
2: {
3:     var transformCache = transform;
4:     transformCache.SetPositionAndRotation(position, rotation);
5:     transformCache.localScale = scale;
6: }
```

9.5 Classes that need to be explicitly discarded

Since Unity is developed in C#, objects that are no longer referenced by GC are freed. However, some classes in Unity need to be explicitly destroyed. Typical examples are `Texture2D`, `Sprite`, `Material`, and `PlayableGraph`. If you generate them with `new` or the dedicated `Create` function, be sure to explicitly destroy them.

▼ List 9.7 Generation and Explicit Destruction

```
1: void Start()
2: {
3:     _texture = new Texture2D(8, 8);
4:     _sprite = Sprite.Create(_texture, new Rect(0, 0, 8, 8), Vector2.zero);
5:     _material = new Material(shader);
6:     _graph = PlayableGraph.Create();
7: }
8:
9: void OnDestroy()
10: {
11:     Destroy(_texture);
12:     Destroy(_sprite);
13:     Destroy(_material);
14:
15:     if (_graph.IsValid())
16:     {
17:         _graph.Destroy();
18:     }
19: }
```

9.6 String specification

Avoid using strings to specify states to play in `Animator` and properties to manipulate in `Material`.

▼ List 9.8 Example of String Specification

Chapter 9 Tuning Practice - Script (Unity)

```
1: _animator.Play("Wait");
2: _material.SetFloat("_Prop", 100f);
```

Inside these functions, `Animator.StringToHash()` and `Shader.PropertyToID()` are executed to convert strings to unique identification values. Since it is wasteful to perform the conversion each time when accessing the site many times, cache the identification value and use it repeatedly. As shown in the sample below, it is recommended to define a class that lists cached identification values for ease of use.

▼ List 9.9 Example of caching identification values

```
1: public static class ShaderProperty
2: {
3:     public static readonly int Color = Shader.PropertyToID("_Color");
4:     public static readonly int Alpha = Shader.PropertyToID("_Alpha");
5:     public static readonly int ZWrite = Shader.PropertyToID("_ZWrite");
6: }
7: public static class AnimationState
8: {
9:     public static readonly int Idle = Animator.StringToHash("idle");
10:    public static readonly int Walk = Animator.StringToHash("walk");
11:    public static readonly int Run = Animator.StringToHash("run");
12: }
```

9.7 Pitfalls of JsonUtility

Unity provides a class `JsonUtility` for JSON serialization/deserialization. The official document ^{*3} also states that it is faster than the C# standard, and is often used for performance-conscious implementations.

`JsonUtility` (although it has less functionality than .NET JSON) has been shown in benchmark tests to be significantly faster than the commonly used .

However, there is one performance-related thing to be aware of. .NET JSON, but there is one performance-related issue to be aware of: the handling of `null`.

The sample code below shows the serialization process and its results. You can see that even though the member `b1` of class `A` is explicitly set to `null`, it is serialized with class `B` and class `C` generated with the default constructor. If the field to be

^{*3} <https://docs.unity3d.com/ja/current/Manual/JSONSerialization.html>

serialized has null as shown here, a dummy object will be new created during JSON conversion, so you may want to take that overhead into account.

▼ List 9.10 Serialization Behavior

```
1: [Serializable] public class A { public B b1; }
2: [Serializable] public class B { public C c1; public C c2; }
3: [Serializable] public class C { public int n; }
4:
5: void Start()
6: {
7:     Debug.Log(JsonUtility.ToJson(new A() { b1 = null, }));
8:     // {"b1":{"c1":{"n":0}, "c2":{"n":0}}
9: }
```

9.8 Pitfalls of Render and MeshFilter

Materials obtained with `Renderer.material` and meshes obtained with `MeshFilter.mesh` are duplicated instances and must be explicitly destroyed when finished using them. The official documentation^{*4*5} also clearly states the following respectively.

If the material is used by any other renderers, this will clone the shared material and start using it from now on.

It is your

Keep acquired materials and meshes in member variables and destroy them at the appropriate time. It is your responsibility to destroy the automatically instantiated mesh when the game object is being destroyed.

▼ List 9.11 Explicitly destroying duplicated materials

```
1: void Start()
2: {
3:     _material = GetComponent<Renderer>().material;
4: }
5:
6: void OnDestroy()
7: {
8:     if (_material != null) {
9:         Destroy(_material);
10:    }
```

^{*4} <https://docs.unity3d.com/ja/current/ScriptReference/Renderer-material.html>

^{*5} <https://docs.unity3d.com/ja/current/ScriptReference/MeshFilter-mesh.html>

```
11: }
```

9.9 Removal of log output codes

Unity provides functions for log output such as `Debug.Log()`, `Debug.LogWarning()`, and `Debug.LogError()`. While these functions are useful, there are some problems with them.

- Log output itself is a heavy process.
- It is also executed in release builds.
- String generation and concatenation causes `GC.Alloc`.

If you turn off the Logging setting in Unity, the stack trace will stop, but the logs will be output. If `UnityEngine.Debug.unityLogger.logEnabled` is set to `false` in Unity, no logging is output, but since it is just a branch inside the function, function call costs and string generation and concatenation that should be unnecessary are done. There is also the option of using the `#if` directive, but it is not realistic to deal with all log output processing.

▼ List 9.12 The `#if` directive

```
1: #if UNITY_EDITOR
2:   Debug.LogError($"Error {e}");
3: #endif
```

The `Conditional` attribute can be utilized in such cases. Functions with the `Conditional` attribute will have the calling part removed by the compiler if the specified symbol is not defined. List 9.13 As in the sample in #1, it is a good idea to add the `Conditional` attribute to each function on the home-made class side as a rule to call the logging function on the Unity side through the home-made log output class, so that the entire function call can be removed if necessary.

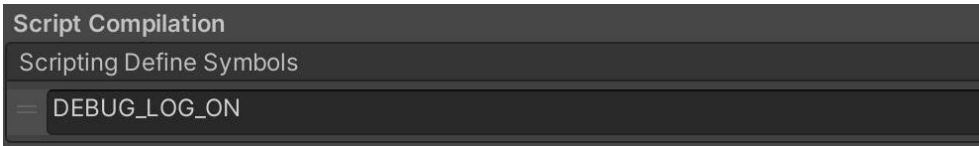
▼ List 9.13 Example of Conditional Attribute

```
1: public static class Debug
2: {
3:     private const string MConditionalDefine = "DEBUG_LOG_ON";
4:
5:     [System.Diagnostics.Conditional(MConditionalDefine)]
```



```
6:     public static void Log(object message)
7:         => UnityEngine.Debug.Log(message);
8: }
```

One thing to note is that the symbols specified must be able to be referenced by the function caller. The scope of the symbols defined in `#define` would be limited to the file in which they are written. It is not practical to define a symbol in every file that calls a function with the `Conditional` attribute. Unity has a feature called Scripting Define Symbols that allows you to define symbols for the entire project. This can be done under "Project Settings -> Player -> Other Settings".



▲ Figure 9.1 Scripting Define Symbols

9.10 Accelerate your code with Burst

Burst ^{*6} is an official Unity compiler for high-performance C# scripting.

Burst uses a subset of the C# language to write code. Burst converts the C# code into IR (Intermediate Representation), which is the intermediate syntax of ^{*7}, a compiler infrastructure called LLVM, and then optimizes the IR before converting it into machine language.

At this point, the code is vectorized as much as possible and replaced with SIMD, a process that actively uses instructions. This is expected to produce faster program output.

SIMD stands for Single Instruction/Multiple Data and refers to instructions that apply a single instruction to multiple data simultaneously. In other words, by actively using SIMD instructions, data is processed together in a single instruction, resulting in faster operation compared to normal instructions.

^{*6} <https://docs.unity3d.com/Packages/com.unity.burst@1.6/manual/docs/QuickStart.html>

^{*7} <https://llvm.org/>

9.10.1 Using Burst to Speed Up Code

Burst uses a subset of C# called High Performance C# (HPC#) ^{*8} to write code.

One of the features of HPC# is that C# reference types, such as classes and arrays, are not available. Therefore, as a rule, data structures are described using structures.

For collections such as arrays, use `NativeContainer` ^{*9} such as `NativeArray<T>` instead. For more details on HPC#, please refer to the documentation listed in the footnote.

Burst is used in conjunction with the C# Job System. Therefore, its own processing is described in the `Execute` method of a job that implements `IJob`. By giving the `BurstCompile` attribute to the defined job, the job will be optimized by Burst.

List 9.14 shows an example of squaring each element of a given array and storing it in the `Output` array.

▼ List 9.14 Job implementation for a simple validation

```
1: [BurstCompile]
2: private struct MyJob : IJob
3: {
4:     [ReadOnly]
5:     public NativeArray<float> Input;
6:
7:     [WriteOnly]
8:     public NativeArray<float> Output;
9:
10:    public void Execute()
11:    {
12:        for (int i = 0; i < Input.Length; i++)
13:        {
14:            Output[i] = Input[i] * Input[i];
15:        }
16:    }
17: }
```

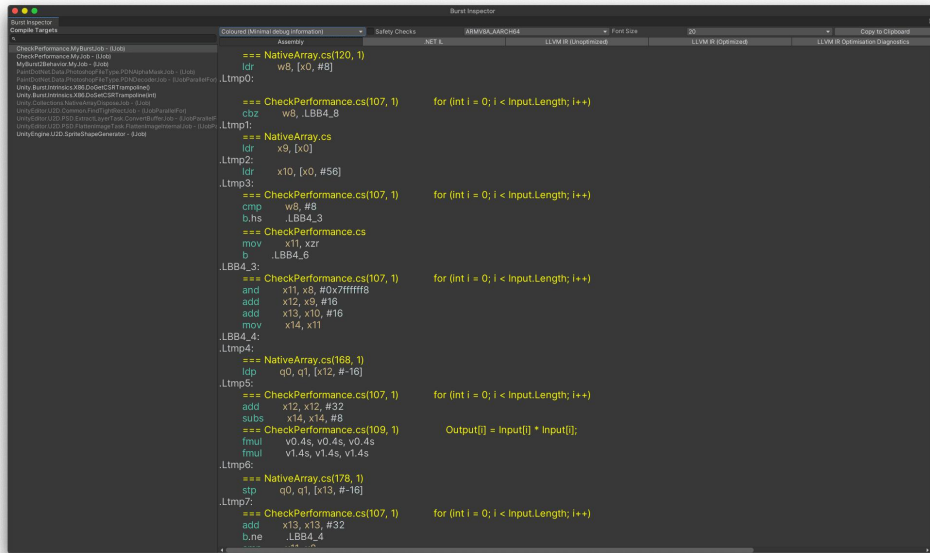
List 9.14 Each element in line 14 of the job can be computed independently (there is no order dependence in the computation), and since the memory alignment of the output array is continuous, they can be computed together using the SIMD instruction.

^{*8} https://docs.unity3d.com/Packages/com.unity.burst@1.7/manual/docs/CSharpLanguageSupport_Types.html

^{*9} <https://docs.unity3d.com/Manual/JobSystemNativeContainer.html>

9.10 Accelerate your code with Burst

You can see what kind of assembly the code will be converted to using Burst Inspector as shown at Figure 9.2.



▲ Figure 9.2 Using the Burst Inspector, you can check what kind of assembly the code will be converted to.

List 9.14 The process on line 14 of the code will be converted to List 9.15 in an assembly for ARMV8A_AARCH64.

▼ List 9.15 List 9.14 Line 14 of the assembly for ARMV8A_AARCH64

```
1:      fmul      v0.4s, v0.4s, v0.4s
2:      fmul      v1.4s, v1.4s, v1.4s
```

The fact that the operand of the assembly is suffixed with `.4s` confirms that the SIMD instruction is used.

The performance of the code implemented with pure C# and the code optimized with Burst are compared on a real device.

The actual devices are Android Pixel 4a and IL2CPP built with a script backend for comparison. The array size is $2^{20} = 1,048,576$. The same process was repeated 10 times and the average processing time was taken.

Chapter 9 Tuning Practice - Script (Unity)

Table 9.1 The results of the performance comparison are shown in Figure 2.

▼ Table 9.1 Comparison of processing time between pure C# implementation and Burst optimized code

Method	Processing time (hidden)
Pure C# implementation	5.73 ms
Implementation with Burst	0.98ms

We observed a speedup of about 5.8 times compared to the pure C# implementation.



PERFORMANCE TUNING BIBLE

CHAPTER

10

第10章

**Tuning Practice
— Script (C#) —**

CyberAgent Smartphone Games & Entertainment

Chapter 10

Tuning Practice - Script (C#)

This chapter mainly introduces performance tuning practices for C# code with examples. Basic C# notation is not covered here, but rather the design and implementation that you should be aware of when developing games that require performance.

10.1 GC.Alloc cases and how to deal with them

As introduced in "2.5.2 Garbage Collection", in this section, let's first understand what kind of specific processing causes GC.Alloc.

10.1.1 New of reference type

First of all, this is a very simple case in which GC.Alloc occurs.

▼ List 10.1 Code that GC.Alloc every frame

```
1: private void Update()
2: {
3:     const int listCapacity = 100;
4:     // GC.Alloc in new of List<int>.
5:     var list = new List<int>(listCapacity);
6:     for (var index = 0; index < listCapacity; index++)
7:     {
8:         // Pack index into list, though it doesn't make any sense in particular
9:         list.Add(index);
10:    }
11:    // Randomly take a value from the list
12:    var random = UnityEngine.Random.Range(0, listCapacity);
13:    var randomValue = list[random];
14:    // ... Do something with the random value ...
15: }
```

The major problem with this code is that `List<int>` is new in the Update method that is executed every frame.

To fix this, it is possible to avoid GC.Alloc every frame by pre-generating `List<int>` and using it around.

10.1 GC.Alloc cases and how to deal with them

▼ List 10.2 Code that eliminates GC.Alloc in every frame

```
1: private static readonly int listCapacity = 100;
2: // Generate a List in advance
3: private readonly List<int> _list = new List<int>(listCapacity);
4:
5: private void Update()
6: {
7:     _list.Clear();
8:     for (var index = 0; index < listCapacity; index++)
9:     {
10:         // Pack indexes into the list, though it doesn't make sense to do so
11:         _list.Add(index);
12:     }
13:     // Randomly take a value from the list
14:     var random = UnityEngine.Random.Range(0, listCapacity);
15:     var randomValue = _list[random];
16:     // ... Do something with the random values ...
17: }
```

I don't think you will ever write meaningless code like the sample code here, but similar examples can be found in more cases than you might imagine.

If you lose GC.Alloc.

As you may have noticed, the sample code from List 10.2 above is all you need to do.

```
1: var randomValue = UnityEngine.Random.Range(0, listCapacity);
2: // ... Do something from a random value ...
```

While it is important to think about eliminating GC.Alloc in performance tuning, always thinking about eliminating pointless calculations is a step toward speeding up the process.

10.1.2 Lambda Expressions

Lambda expressions are also a useful feature, but their use is limited in games because they too can cause GC.Alloc depending on how they are used. Here we assume that the following code is defined.

Chapter 10 Tuning Practice - Script (C#)

▼ List 10.4 Assumed code for the lambda expression sample

```
1: // Member Variables
2: private int _memberCount = 0;
3:
4: // static variables
5: private static int _staticCount = 0;
6:
7: // member method
8: private void IncrementMemberCount()
9: {
10:     _memberCount++;
11: }
12:
13: // static method
14: private static void IncrementStaticCount()
15: {
16:     _staticCount++;
17: }
18:
19: // Member method that only invokes the received Action
20: private void InvokeActionMethod(System.Action action)
21: {
22:     action.Invoke();
23: }
```

At this time, if a variable is referenced in a lambda expression as follows, GC.Alloc will occur.

▼ List 10.5 Case of GC.Alloc by referencing a variable in a lambda expression

```
1: // When a member variable is referenced, Delegate Allocation occurs
2: InvokeActionMethod(() => { _memberCount++; });
3:
4: // When a local variable is referenced, Closure Allocation occurs
5: int count = 0;
6: // The same Delegate Allocation as above also occurs
7: InvokeActionMethod(() => { count++; });
```

However, these GC.Alloc can be avoided by referencing static variables as follows.

▼ List 10.6 Cases where a static variable is referenced in a lambda expression and GC.Alloc is not performed

```
1: // When a static variable is referenced, GC Alloc does not occur and
2: InvokeActionMethod(() => { _staticCount++; });
```

GC.Alloc is also performed differently for method references in lambda expressions, depending on how they are written.

10.1 GC.Alloc cases and how to deal with them

▼ List 10.7 Cases of GC.Alloc when a method is referenced in a lambda expression

```
1: // When a member method is referenced, Delegate Allocation occurs.
2: InvokeActionMethod(() => { IncrementMemberCount(); });
3:
4: // If a member method is directly specified, Delegate Allocation occurs.
5: InvokeActionMethod(IncrementMemberCount);
6:
7: // When a static method is directly specified, Delegate Allocation occurs
8: InvokeActionMethod(IncrementStaticCount);
```

To avoid these cases, it is necessary to reference static methods in a statement format as follows.

▼ List 10.8 Cases where a method is referenced in a lambda expression and GC.Alloc is not performed

```
1: // Non Alloc when a static method is referenced in a lambda expression
2: InvokeActionMethod(() => { IncrementStaticCount(); });
```

In this way, the Action is new only the first time, but it is cached internally to avoid GC.Alloc from the second time onward.

However, making all variables and methods static is not very adoptable in terms of code safety and readability. In code that needs to be fast, it is safer to design without using lambda expressions for events that fire at every frame or at indefinite times, rather than to use a lot of statics to eliminate GC.Alloc.

10.1.3 Cases where generics are used and boxed

In the following cases where generics are used, what could cause boxing?

▼ List 10.9 Example of possible boxed cases using generics

```
1: public readonly struct GenericStruct<T> : IEquatable<T>
2: {
3:     private readonly T _value;
4:
5:     public GenericStruct(T value)
6:     {
7:         _value = value;
8:     }
9:
10:    public bool Equals(T other)
11:    {
12:        var result = _value.Equals(other);
```

Chapter 10 Tuning Practice - Script (C#)

```
13:         return result;
14:     }
15: }
```

In this case, the programmer implemented the `IEquatable<T>` interface to `GenericStruct`, but forgot to place restrictions on `T`. As a result, a type that does not implement the `IEquatable<T>` interface can be specified for `T`, and there is a case where the following `Equals` is used by implicitly casting to the `Object` type.

▼ List 10.10 Object.cs

```
1: public virtual bool Equals(object obj);
```

For example, if `struct`, which does not implement the `IEquatable<T>` interface, is specified to `T`, it will be cast to `object` with the argument `Equals`, resulting in boxing. To prevent this from happening in advance, change the following

▼ List 10.11 Example with restrictions to prevent boxing

```
1: public readonly struct GenericOnlyStruct<T> : IEquatable<T>
2:     where T : IEquatable<T>
3: {
4:     private readonly T _value;
5:
6:     public GenericOnlyStruct(T value)
7:     {
8:         _value = value;
9:     }
10:
11:     public bool Equals(T other)
12:     {
13:         var result = _value.Equals(other);
14:         return result;
15:     }
16: }
```

By using the `where` clause (generic type constraint) to restrict the types that `T` can accept to those that implement `IEquatable<T>`, such unexpected boxing can be prevented.

Never lose sight of the original purpose

As introduced in "2.5.2 Garbage Collection", there are many cases where the structure is chosen because the intention is to avoid GC.Alloc during run-time in games. However, it is not always possible to speed up the process by making everything a structure in order to reduce GC.Alloc.

One of the most common failures is that when structs are used to avoid GC.Alloc, the cost related to GC is reduced as expected, but the data size is so large that copying the value type becomes expensive, resulting in inefficient processing.

To avoid this, there are also methods that reduce copying costs by using pass-by-reference for method arguments. Although this may result in a speed-up, in this case, you should consider selecting a class from the beginning and implementing it in such a way that instances are pre-generated and used around. Remember that the ultimate goal is not to eradicate GC.Alloc, but to reduce the processing time per frame.

10.2 About for/foreach

As introduced in "2.6 Algorithms and computational complexity", loops become time-consuming depending on the number of data. Also, loops, which at first glance appear to be the same process, can vary in efficiency depending on how the code is written.

Let's take a look at the results of decompiling the code from IL to C# using SharpLab ^{*1}, using `foreach/for List` and just getting the contents of the array one by one.

First, let's look at the loop around `foreach`. `List` I've omitted adding values to the

▼ List 10.12 Example of looping through a List with foreach

```
1: var list = new List<int>(128);  
2: foreach (var val in list)  
3: {  
4: }
```

^{*1} <https://sharplab.io/>

Chapter 10 Tuning Practice - Script (C#)

▼ List 10.13 Decompilation result of the example of looping through a List with foreach

```
1: List<int>.Enumerator enumerator = new List<int>(128).GetEnumerator();
2: try
3: {
4:     while (enumerator.MoveNext())
5:     {
6:         int current = enumerator.Current;
7:     }
8: }
9: finally
10: {
11:     ((IDisposable)enumerator).Dispose();
12: }
```

In the case of turning with `foreach`, you can see that the implementation is to get the enumerator, move on with `MoveNext()`, and refer to the value with `Current`. Furthermore, looking at the implementation of `MoveNext()` in `list.cs`^{*2}, it appears that the number of various property accesses, such as size checks, are increased, and that processing is more frequent than direct access by the indexer.

Next, let's look at when we turn in `for`.

▼ List 10.14 Example of turning a List with for

```
1: var list = new List<int>(128);
2: for (var i = 0; i < list.Count; i++)
3: {
4:     var val = list[i];
5: }
```

▼ List 10.15 Decompiled result when turning List with for

```
1: List<int> list = new List<int>(128);
2: int num = 0;
3: while (num < list.Count)
4: {
5:     int num2 = list[num];
6:     num++;
7: }
```

In C#, the `for` statement is a sugar-coated syntax for the `while` statement, and the indexer (`public T this[int index]`), and is obtained by reference by the indexer (Also, if you look closely at this `while` statement, you will see that the conditional

^{*2} <https://referencesource.microsoft.com/#mscorlib/system/collections/generic/list.cs>

expression contains `list.Count`. This means that the access to the `Count` property is performed each time the loop is repeated. `Count` The more the number of `Count` accesses to the property, the more the number of accesses to the property increases proportionally, and depending on the number of accesses, the load becomes non-negligible. If `Count` does not change within the loop, then the load on property accesses can be reduced by caching them before the loop.

▼ List 10.16 Example of turning a List with for: Improved version

```
1: var count = list.Count;
2: for (var i = 0; i < count; i++)
3: {
4:     var val = list[i];
5: }
```

▼ List 10.17 Example of List in for: Decompiled result of the improved version

```
1: List<int> list = new List<int>(128);
2: int count = list.Count;
3: int num = 0;
4: while (num < count)
5: {
6:     int num2 = list[num];
7:     num++;
8: }
```

Caching `Count` reduced the number of property accesses and made it faster. Both of the comparisons in this loop are not loaded by `GC.Alloc`, and the difference is due to the difference in implementation.

In the case of arrays, `foreach` has also been optimized and is almost unchanged from that described in `for`.

▼ List 10.18 Example of turning an array with foreach

```
1: var array = new int[128];
2: foreach (var val in array)
3: {
4: }
```

▼ List 10.19 Decompilation result of the example of turning an array by foreach

```
1: int[] array = new int[128];
2: int num = 0;
3: while (num < array.Length)
4: {
5:     int num2 = array[num];
6:     num++;
7: }
```

For the purpose of verification, the number of data is 10,000,000 and random numbers are assigned in advance. `List<int>` The sum of the data is calculated. The verification environment was Pixel 3a and Unity 2021.3.1f1.

▼ Table 10.1 Measurement results for each description method in `List<int>`

Type	Time ms
List: foreach	66.43
List: for	62.49
List: for (Count cache)	55.11
Array: for	30.53
Array: foreach	23.75

In the case of `List<int>`, a comparison with a finer set of conditions shows that `foreach` and `for` with `Count` optimizations are even faster than `foreach`. `List` The `foreach` can be rewritten to `for` with `Count` optimization to reduce the overhead of the `MoveNext()` and `Current` properties in the processing of `foreach`, thus making it faster.

In addition, when comparing the respective fastest speeds of `List` and arrays, arrays are approximately 2.3 times faster than `List`. Even if `foreach` and `for` are written to have the same IL result, `foreach` is the faster result, and array's `foreach` is sufficiently optimized.

Based on the above results, arrays should be considered instead of `List<T>` for situations where the number of data is large and processing speed must be fast.

However, if the rewriting is insufficient, such as when `List` defined in a field is referenced without local caching, it may not be possible to speed up the process.

10.3 Object Pooling

As we have mentioned in many places, it is important in game development to pre-generate objects and use them around instead of dynamically generating them. This

is called **object pooling**. For example, objects that are to be used in the game phase can be pooled together in the load phase and handled while only assigning and referencing the pooled objects when they are used, thereby avoiding GC.Alloc during the game phase.

In addition to reducing allocations, object pooling can also be used in a variety of other situations, such as enabling screen transitions without having to recreate the objects that make up the screen each time, reducing load times, and avoiding multiple heavy calculations by retaining the results of processes with very high calculation costs. It is used in a variety of situations.

Although the term "object" is used here in a broad sense, it applies not only to the smallest unit of data, but also to `Coroutine` and `Action`, . For example, consider generating `Coroutine` more than the expected number of executions in advance, and use it when necessary to exhaust it. For example, if a game that takes 2 minutes to complete will be executed a maximum of 20 times, you can reduce the cost of generating by generating `IEnumerator` in advance and only using `StartCoroutine` when you need to use it.

10.4 string

The `string` object is a sequential collection of `System.Char` objects representing strings. `string GC.Alloc` can easily occur with one usage. For example, concatenating two strings using the character concatenation operator `+` will result in the creation of a new `string` object. `string` The value of cannot be changed (immutable) after it is created, so an operation that appears to change the value creates and returns a new `string` object.

▼ List 10.20 When string concatenation is used to create a `STRING`

```
1: private string CreatePath()
2: {
3:     var path = "root";
4:     path += "/";
5:     path += "Hoge";
6:     path += "/";
7:     path += "Fuga";
8:     return path;
9: }
```

In the above example, a string is created with each string concatenation, resulting

in a total of 164Byte allocation.

When strings are frequently changed, the use of `StringBuilder`, whose value can be changed, can prevent the mass generation of `string` objects. By performing operations such as character concatenation and deletion in the `StringBuilder` object and finally extracting the value and `ToString()` it to the `string` object, the memory allocation can be limited to only the time of acquisition. Also, when using `StringBuilder`, be sure to set `Capacity`. When unspecified, the default value is 16, and when the buffer is extended with more characters, such as `Append`, memory allocation and value copying will run. Be sure to set an appropriate `Capacity` that will not cause inadvertent expansion.

▼ List 10.21 When creating a string with `StringBuilder`

```
1: private readonly StringBuilder _stringBuilder = new StringBuilder(16);
2: private string CreatePathFromStringBuilder()
3: {
4:     _stringBuilder.Clear();
5:     _stringBuilder.Append("root");
6:     _stringBuilder.Append("/");
7:     _stringBuilder.Append("Hoge");
8:     _stringBuilder.Append("/");
9:     _stringBuilder.Append("Fuga");
10:    return _stringBuilder.ToString();
11: }
```

In the example using `StringBuilder`, if `StringBuilder` is generated in advance (in the above example, 112Byte allocation is made at the time of generation), then from onward, only 50Byte allocation is needed which is taken at `ToString()` when the generated string is retrieved.

However, `StringBuilder` is also not recommended for use when you want to avoid `GC.Alloc`, since allocation is only less likely to occur during value manipulation, and as mentioned above, `string` objects will be generated when `ToString()` is executed. Also, since the `$""` syntax is converted to `string.Format` and the internal implementation of `string.Format` uses `StringBuilder`, the cost of `ToString()` is ultimately unavoidable. The use of objects in the previous section should be applied here as well, and strings that may be used in advance should be pre-generated `string` objects and used.

However, there are times during the game when string manipulation and the creation of `string` objects must be performed. In such cases, it is necessary to have a pre-generated buffer for strings and extend it so that it can be used as is. Consider

implementing your own code like `unsafe` or introducing a library with extensions for Unity like `ZString` ^{*3} (e.g. `NonAlloc` applicability to `TextMeshPro`).

10.5 LINQ and Latency Evaluation

This section describes how to reduce `GC.Alloc` by using LINQ and the key points of lazy evaluation.

Mitigating `GC.Alloc` by using LINQ

The use of LINQ causes `GC.Alloc` in cases like List 10.22.

▼ List 10.22 Example of `GC.Alloc` occurring

```
1: var oneToTen = Enumerable.Range(1, 11).ToArray();
2: var query = oneToTen.Where(i => i % 2 == 0).Select(i => i * i);
```

List 10.22 The reason why `GC.Alloc` occurs in is due to the internal implementation of LINQ. In addition, some LINQ methods are optimized for the caller's type, so the size of `GC.Alloc` changes depending on the caller's type.

▼ List 10.23 Execution speed verification for each type

```
1: private int[] array;
2: private List<int> list;
3: private IEnumerable<int> ienumerable;
4:
5: public void GlobalSetup()
6: {
7:     array = Enumerable.Range(0, 1000).ToArray();
8:     list = Enumerable.Range(0, 1000).ToList();
9:     ienumerable = Enumerable.Range(0, 1000);
10: }
11:
12: public void RunAsArray()
13: {
14:     var query = array.Where(i => i % 2 == 0);
15:     foreach (var i in query){}
16: }
17:
18: public void RunAsList()
19: {
20:     var query = list.Where(i => i % 2 == 0);
21:     foreach (var i in query){}
22: }
```

^{*3} <https://github.com/Cysharp/ZString>

```
23:
24: public void RunAsIEnumerable()
25: {
26:     var query = ienumerable.Where(i => i % 2 == 0);
27:     foreach (var i in query){}
28: }
```

List 10.23 We measured the benchmark for each method defined in Figure 10.1. The results show that the size of heap allocations increases in the order `T[]` → `List<T>` → `IEnumerable<T>`.

Thus, when using LINQ, the size of GC.Alloc can be reduced by being aware of the runtime type.

Method	Mean	Error	StdDev	Ratio	RatioSD	Allocated
RunFromArray	4.210 us	0.2735 us	0.0150 us	1.00	0.00	48 B
RunAsList	4.942 us	0.2517 us	0.0138 us	1.17	0.00	72 B
RunAsIEnumerable	7.326 us	3.4885 us	0.1912 us	1.74	0.04	96 B

▲ Figure 10.1 Comparison of Execution Speed by Type

Causes of GC.Alloc in LINQ

Part of the cause of GC.Alloc with the use of LINQ is the internal implementation of LINQ. Many LINQ methods take `IEnumerable<T>` and return `IEnumerable<T>`, and this API design allows for intuitive description using method chains. The entity `IEnumerable<T>` returned by a method is an instance of the class for each function. LINQ internally instantiates a class that implements `IEnumerable<T>`, and furthermore, GC.Alloc occurs internally because calls to `GetEnumerator()` are made to realize loop processing, etc.

LINQ Lazy Evaluation

LINQ methods such as `Where` and `Select` are lazy evaluations that delay evaluation until the result is actually needed. On the other hand, methods such as `ToArray` are defined for immediate evaluation.

Now consider the case of the following List 10.24 code.

▼ List 10.24 Methods with immediate evaluation in between

```
1: private static void LazyExpression()
2: {
3:     var array = Enumerable.Range(0, 5).ToArray();
4:     var sw = Stopwatch.StartNew();
5:     var query = array.Where(i => i % 2 == 0).Select(HeavyProcess).ToArray();
6:     Console.WriteLine($"Query: {sw.ElapsedMilliseconds}");
7:
8:     foreach (var i in query)
9:     {
10:         Console.WriteLine($"diff: {sw.ElapsedMilliseconds}");
11:     }
12: }
13:
14: private static int HeavyProcess(int x)
15: {
16:     Thread.Sleep(1000);
17:     return x;
18: }
```

List 10.24 The result of the execution of List 10.25 is the result of . By adding `ToArray` at the end, which is an immediate evaluation, the result of executing the method `Where` or `Select` and evaluating the value is returned when the assignment is made to `query`. Therefore, since `HeavyProcess` is also called, you can see that processing time is taken at the timing when `query` is generated.

▼ List 10.25 Result of Adding a Method for Immediate Evaluation

```
1: Query: 3013
2: diff: 3032
3: diff: 3032
4: diff: 3032
```

As you can see, unintentional calls to LINQ's immediate evaluation methods can result in bottlenecks at those points. `ToArray` Methods that require looking at the entire sequence once, such as `OrderBy`, `Count`, and `ToArray`, are immediate evaluation, so be aware of the cost when calling them.

The Choice to "Avoid Using LINQ"

This section explained the causes of `GC.Alloc` when using LINQ, how to reduce it, and the key points of delayed evaluation. In this section, we explain the criteria for using LINQ. The premise is that LINQ is a useful language feature, but its use will

Chapter 10 Tuning Practice - Script (C#)

worsen heap allocation and execution speed compared to when it is not used. In fact, Microsoft's Unity performance recommendations at ^{*4} clearly state "Avoid use of LINQ. Here is a benchmark comparison of the same logic implementation with and without LINQ at List 10.26.

▼ List 10.26 Performance comparison with and without LINQ

```
1: private int[] array;
2:
3: public void GlobalSetup()
4: {
5:     array = Enumerable.Range(0, 100_000_000).ToArray();
6: }
7:
8: public void Pure()
9: {
10:     foreach (var i in array)
11:     {
12:         if (i % 2 == 0)
13:         {
14:             var _ = i * i;
15:         }
16:     }
17: }
18:
19: public void UseLinq()
20: {
21:     var query = array.Where(i => i % 2 == 0).Select(i => i * i);
22:     foreach (var i in query)
23:     {
24:     }
25: }
```

The results are available at Figure 10.2. The comparison of execution times shows that the process with LINQ takes 19 times longer than the process without LINQ.

Method	Mean	Error	StdDev	Ratio	RatioSD	Allocated
Pure	26.06 ms	3.230 ms	0.177 ms	1.00	0.00	26 B
UseLinq	514.55 ms	354.586 ms	19.436 ms	19.75	0.79	920 B

▲ Figure 10.2 Performance Comparison Results with and without LINQ

While the above results clearly show that the use of LINQ deteriorates performance, there are cases where the coding intent is more easily conveyed by using

^{*4} <https://docs.microsoft.com/en-us/windows/mixed-reality/develop/unity/performance-recommendations-for-unity#avoid-expensive-operations>

LINQ. After understanding these behaviors, there may be room for discussion within the project as to whether to use LINQ or not, and if so, the rules for using LINQ.

10.6 How to avoid async/await overhead

Async/await is a language feature added in C# 5.0 that allows asynchronous processing to be written as a single synchronous process without callbacks.

Avoid async where it is not needed

Methods defined async will have code generated by the compiler to achieve asynchronous processing. And if the async keyword is present, code generation by the compiler is always performed. Therefore, even methods that may complete synchronously, such as List 10.27, are actually code generated by the compiler.

▼ List 10.27 Asynchronous processing that may complete synchronously

```
1: using System;
2: using System.Threading.Tasks;
3:
4: namespace A {
5:     public class B {
6:         public async Task HogeAsync(int i) {
7:             if (i == 0) {
8:                 Console.WriteLine("i is 0");
9:                 return;
10:            }
11:            await Task.Delay(TimeSpan.FromSeconds(1));
12:        }
13:
14:        public void Main() {
15:            int i = int.Parse(Console.ReadLine());
16:            Task.Run(() => HogeAsync(i));
17:        }
18:    }
19: }
```

In cases such as List 10.27, the cost of generating a state machine structure for `IAsyncStateMachine` implementation, which is unnecessary in the case of synchronous completion, can be omitted by splitting `HogeAsync`, which may be completed synchronously, and implementing it as List 10.28.

▼ List 10.28 Split implementation of synchronous and asynchronous processing

```
1: using System;
2: using System.Threading.Tasks;
3:
4: namespace A {
5:     public class B {
6:         public async Task HogeAsync(int i) {
7:             await Task.Delay(TimeSpan.FromSeconds(1));
8:         }
9:
10:        public void Main() {
11:            int i = int.Parse(Console.ReadLine());
12:            if (i == 0) {
13:                Console.WriteLine("i is 0");
14:            } else {
15:                Task.Run(() => HogeAsync(i));
16:            }
17:        }
18:    }
19: }
```

How async/await works

The async/await syntax is realized using compiler code generation at compile time. Methods with the async keyword add a process to generate a structure implementing `IAsyncStateMachine` at compile time, and the async/await function is realized by managing a state machine that advances state when the process to be awaited completes. Also, this `IAsyncStateMachine` is an interface defined in the `System.Runtime.CompilerServices` namespace and is available only to the compiler.

Avoid capturing synchronous context

The mechanism to return to the calling thread from asynchronous processing that has been saved to another thread is synchronous context, and `await` The previous context can be captured by using `CaptureSyncContext`. Since this synchronous context is captured each time `await` is executed, there is an overhead for each `await`. For this reason, `UniTask`^{*5}, which is widely used in Unity development, is implemented without `ExecutionContext` and `SynchronizationContext` to avoid the overhead of synchronous context

^{*5} <https://tech.cygames.co.jp/archives/3417/>

capture. As far as Unity is concerned, implementing such libraries may improve performance.

10.7 Optimization with `stackalloc`

Allocating arrays as local variables causes GC.Alloc to occur each time, which can lead to spikes. In addition, reading and writing to the heap area is a little less efficient than to the stack area.

Therefore, in C#, the **unsafe** code-only syntax for allocating arrays on the stack.

List 10.29 Instead of using the `new` keyword, as in the following example, an array can be allocated on the stack using the `stackalloc` keyword.

▼ List 10.29 `stackalloc` Allocating an array on the stack using the

```
1: // stackalloc is limited to unsafe
2: unsafe
3: {
4:     // Allocating an array of ints on the stack
5:     byte* buffer = stackalloc byte[BufferSize];
6: }
```

Since C# 7.2, the `Span<T>` structure can be used to allocate an array of ints on the stack as shown in List 10.30. **The structure can now be used without unsafe** `stackalloc` can be used without `unsafe` as shown in .

▼ List 10.30 `Span<T>` Allocating an array on the stack using the struct

```
1: Span<byte> buffer = stackalloc byte[BufferSize];
```

For Unity, this is standard from 2021.2. For earlier versions, `Span<T>` does not exist, so `System.Memory.dll` must be installed.

Arrays allocated with `stackalloc` are stack-only and cannot be held in class or structure fields. They must be used as local variables.

Even though the array is allocated on the stack, it takes a certain amount of processing time to allocate an array with a large number of elements. If you want to use arrays with a large number of elements in places where heap allocation should be avoided, such as in an update loop, it is better to allocate the array in advance during initialization or to prepare a data structure like an object pool, and implement it in such a way that it can be rented out when used.

Also, note that the stack area allocated by `stackalloc` is **not released until the function exits**. For example, the code shown at List 10.31 may cause a Stack Overflow while looping, since all arrays allocated in the loop are retained and released when exiting the `Hoge` method.

▼ List 10.31 `stackalloc` Allocating Arrays on the Stack Using

```
1: unsafe void Hoge()
2: {
3:     for (int i = 0; i < 10000; i++)
4:     {
5:         // Arrays are accumulated for the number of loops
6:         byte* buffer = stackalloc byte[10000];
7:     }
8: }
```

10.8 Optimizing method invocation under IL2CPP backend with sealed

When building with IL2CPP as a backend in Unity, method invocation is performed using a C++ vtable-like mechanism to achieve virtual method invocation of the class^{*6}.

Specifically, for each method call definition of a class, the code shown at List 10.32 is automatically generated.

▼ List 10.32 C++ code for method calls generated by IL2CPP

```
1: struct VirtActionInvoker0
2: {
3:     typedef void (*Action)(void*, const RuntimeMethod*);
4:
5:     static inline void Invoke (
6:         Il2CppMethodSlot slot, RuntimeObject* obj)
7:     {
8:         const VirtualInvokeData& invokeData =
9:             il2cpp_codegen_get_virtual_invoke_data(slot, obj);
10:        ((Action)invokeData.methodPtr)(obj, invokeData.method);
11:    }
12: };
```

It generates **similar** C++ code not only for virtual methods, but also for **non-virtual**

^{*6} <https://blog.unity.com/technology/il2cpp-internals-method-calls>

10.8 Optimizing method invocation under IL2CPP backend with sealed

methods that do not inherit at compile time. This auto-generated behavior leads to **bloated code size and increased processing time for method calls.**

This problem can be avoided by adding the `sealed` modifier to the class definition^{*7}.

List 10.33 If you define a class like List 10.34 and call a method, the C++ code generated by IL2CPP will generate method calls like .

▼ List 10.33 Class definition and method invocation without sealed

```
1: public abstract class Animal
2: {
3:     public abstract string Speak();
4: }
5:
6: public class Cow : Animal
7: {
8:     public override string Speak() {
9:         return "Moo";
10:    }
11: }
12:
13: var cow = new Cow();
14: // Calling the Speak method
15: Debug.LogFormat("The cow says '{0}'", cow.Speak());
```

▼ List 10.34 List 10.33 The C++ code corresponding to the method call in

```
1: // var cow = new Cow();
2: Cow_t1312235562 * L_14 =
3:     (Cow_t1312235562 *)il2cpp_codegen_object_new(
4:         Cow_t1312235562_il2cpp_TypeInfo_var);
5: Cow_ctor_m2285919473(L_14, /* hidden argument*/NULL);
6: V_4 = L_14;
7: Cow_t1312235562 * L_16 = V_4;
8:
9: // cow.Speak()
10: String_t* L_17 = VirtFuncInvoker0< String_t* >::Invoke(
11:     4 /* String AssemblyCSharp.Cow::Speak() */, L_16);
```

List 10.34 shows that `VirtFuncInvoker0< String_t* >::Invoke` is called even though it is not a virtual method call, and that a method call like a virtual method is made.

On the other hand, defining the `Cow` class of List 10.33 with the `sealed` modifier as shown in List 10.35 generates C++ code like List 10.36.

^{*7} <https://blog.unity.com/technology/il2cpp-optimizations-devirtualization>

Chapter 10 Tuning Practice - Script (C#)

▼ List 10.35 Class Definition and Method Calls Using the SEALED

```
1: public sealed class Cow : Animal
2: {
3:     public override string Speak() {
4:         return "Moo";
5:     }
6: }
7:
8: var cow = new Cow();
9: // Calling the Speak method
10: Debug.LogFormat("The cow says '{0}'", cow.Speak());
```

▼ List 10.36 List 10.35 C++ code corresponding to a method call of

```
1: // var cow = new Cow();
2: Cow_t1312235562 * L_14 =
3:     (Cow_t1312235562 *)il2cpp_codegen_object_new(
4:         Cow_t1312235562_il2cpp_TypeInfo_var);
5: Cow__ctor_m2285919473(L_14, /* hidden argument*/NULL);
6: V_4 = L_14;
7: Cow_t1312235562 * L_16 = V_4;
8:
9: // cow.Speak()
10: String_t* L_17 = Cow_Speak_m1607867742(L_16, /* hidden argument*/NULL);
```

Thus, we can see that the method call calls `Cow_Speak_m1607867742`, which directly calls the method.

However, in relatively recent Unity, the Unity official clarifies that such optimization is partially automatic^{*8}.

In other words, even if you do not explicitly specify `sealed`, it is possible that such optimization is done automatically.

However, the "[il2cpp] Is 'sealed' Not Worked As Said Anymore In Unity 2018.3?"

^{*8} As mentioned in the forum, this implementation is not complete as of April 2019.

Because of this current state of affairs, it would be a good idea to check the code generated by IL2CPP and decide on the setting of the `sealed` modifier for each project.

For more reliable direct method calls, and in anticipation of future IL2CPP optimizations, it may be a good idea to set the `sealed` modifier as an optimizable mark.

^{*8} <https://forum.unity.com/threads/il2cpp-is-sealed-not-worked-as-said-anymore-in-unity-2018-3.659017/#post-4412785>

10.9 Optimization through inlining

Method calls have some cost. Therefore, as a general optimization, not only for C# but also for other languages, relatively small method calls are optimized by compilers through inlining.

Specifically, for code such as List 10.37, inlining generates code such as List 10.38.

▼ List 10.37 Code before inlining

```
1: int F(int a, int b, int c)
2: {
3:     var d = Add(a, b);
4:     var e = Add(b, c);
5:     var f = Add(d, e);
6:
7:     return f;
8: }
9:
10: int Add(int a, int b) => a + b;
```

▼ List 10.38 List 10.37 Code with inlining for

```
1: int F(int a, int b, int c)
2: {
3:     var d = a + b;
4:     var e = b + c;
5:     var f = d + e;
6:
7:     return f;
8: }
```

Inlining is done by copying and expanding the contents within a method, such as List 10.38, and the call to the `Add` method within the `Func` method of List 10.37.

In IL2CPP, no particular inlining optimization is performed during code generation.

However, starting with Unity 2020.2, by specifying the `MethodImpl` attribute for a method and `MethodOptions.AggressiveInlining` for its parameter, the corresponding function in the generated C++ code will be given the `inline` specifier. In other words, inlining at the C++ code level is now possible.

The advantage of inlining is that it not only reduces the cost of method calls, but also saves copying of arguments specified at the time of method invocation.

For example, arithmetic methods take multiple relatively large structures as arguments, such as `Vector3` and `Matrix`. If the structs are passed as arguments as they are, they are all copied and passed to the method as passed by value. If the number of arguments and the size of the passed structs are large, the processing cost may be considerable for method calls and argument copying. In addition, method calls may become a case that cannot be overlooked as a processing burden because they are often used in periodic processing, such as in the implementation of physical operations and animations.

In such cases, optimization through inlining can be effective. In fact, Unity's new mathematics library **Mathematics** specifies `MethodOptions.AggressiveInlining` for method calls everywhere^{*9}.

On the other hand, inlining has the disadvantage that the code size increases with the expansion of the process within the method.

Therefore, it is recommended to consider inlining especially for methods that are frequently called in a single frame and are hot-passed. It should also be noted that specifying an attribute does not always result in inlining.

Inlining is limited to methods that are small in content, so methods that you want to inline must be kept small.

Also, in Unity 2020.2 and earlier, the `inline` specifier is not attached to attribute specifications, and there is no guarantee that inlining will be performed reliably even if the C++ `inline` specifier is specified.

Therefore, if you want to ensure inlining, you may want to consider manual inlining for methods that are hotpaths, although it will reduce readability.

^{*9} <https://github.com/Unity-Technologies/Unity.Mathematics/blob/f476dc88954697f71e5615b5f57462495bc973a7/src/Unity.Mathematics/math.cs#L1894>



PERFORMANCE TUNING BIBLE

CHAPTER

11

第11章

**Tuning Practice
— Player Settings —**

CyberAgent Smartphone Games & Entertainment

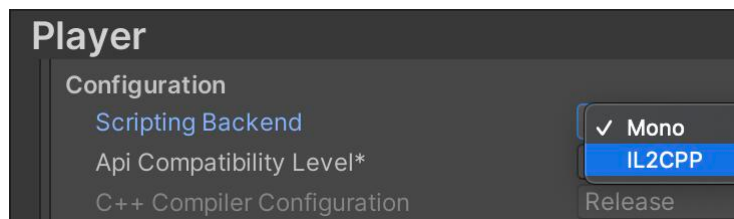
Chapter 11

Tuning Practice - Player Settings

This chapter introduces the Player items in Project Settings that affect performance.

11.1 Scripting Backend

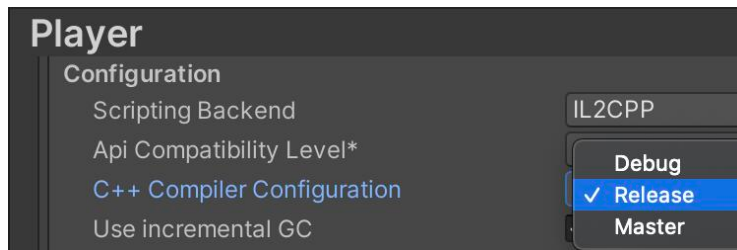
Unity allows you to choose between Mono and IL2CPP as the Scripting Backend on platforms such as Android and Standalone (Windows, macOS, Linux). We recommend choosing IL2CPP because of the performance gains as described in "IL2CPP" of Chapter 2 "Fundamentals".



▲ Figure 11.1 Configuring Scripting Backend

In addition, changing the Scripting Backend to IL2CPP will also change the **C++ Compiler Configuration** can be selected.

11.2 Strip Engine Code / Managed Stripping Level



▲ Figure 11.2 Setting of C++ Compiler Configuration

Here you can choose between Debug, Release, and Master, each of which has a tradeoff between build time and degree of optimization, so it is best to use the one that best suits your build objectives.

11.1.1 Debug

Debug does not perform well at runtime because no optimization is performed, but build time is the shortest compared to the other settings.

11.1.2 Release

Optimization improves run-time performance and reduces the size of built binaries, but increases build time.

11.1.3 Master

All optimizations available for the platform are enabled. For example, Windows builds will use more aggressive optimizations such as link-time code generation (LTCG). In return, build times will be even longer than with the Release setting, but Unity recommends using the Master setting for production builds if this is acceptable.

11.2 Strip Engine Code / Managed Stripping Level

Strip Engine Code is a Unity feature that allows you to set the **Managed Stripping Level** from the CIL bytecode generated by compiling C#, and is expected to reduce the size of the built binary by removing unused code, respectively.

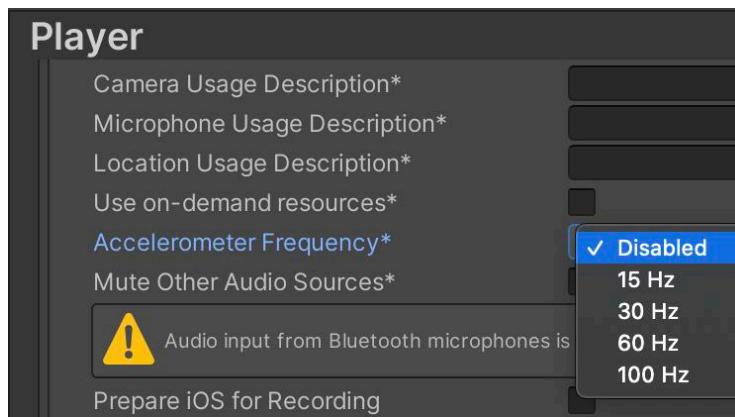
However, since the determination of whether a given code is used relies strongly

on static analysis, types that are not directly referenced in the code, or code that is dynamically called in reflection, may be mistakenly removed.

In such cases, the **link.xml** file or by specifying the `Preserve` attribute. ^{*1}

11.3 Accelerometer Frequency (iOS)

This is an iOS-specific setting that allows you to change the sampling frequency of the accelerometer. The default setting is 60 Hz, so set the frequency appropriately. If you are not using the accelerometer, be sure to disable the setting.



▲ Figure 11.3 Sampling Frequency Setting

^{*1} <https://docs.unity3d.com/2020.3/Documentation/Manual/ManagedCodeStripping.html>



PERFORMANCE TUNING BIBLE

CHAPTER

12

第12章

**Tuning Practice
— Third Party —**

CyberAgent Smartphone Games & Entertainment

Chapter 12

Tuning Practice - Third Party

This chapter introduces some things to keep in mind from a performance perspective when implementing third-party libraries that are often used when developing games in Unity.

12.1 DOTween

DOTween^{*1} is a library that allows scripts to create smooth animations. For example, an animation that zooms in and out can be easily written as the following code

▼ List 12.1 Example of DOTween usage

```
1: public class Example : MonoBehaviour {
2:     public void Play() {
3:         DOTween.Sequence()
4:             .Append(transform.DOScale(Vector3.one * 1.5f, 0.25f))
5:             .Append(transform.DOScale(Vector3.one, 0.125f));
6:     }
7: }
```

12.1.1 SetAutoKill

Since the process of creating a tween, such as `DOTween.Sequence()` or `transform.DOScale(...)`, basically involves memory allocation, consider reusing instances for animations that are frequently replayed.

By default, the tween is automatically discarded when the animation completes, so `SetAutoKill(false)` suppresses this. The first use case can be replaced with the following code

▼ List 12.2 Reusing Tween Instances

^{*1} <http://dotween.demigiant.com/index.php>

```

1:     private Tween _tween;
2:
3:     private void Awake() {
4:         _tween = DOTween.Sequence()
5:             .Append(transform.DOScale(Vector3.one * 1.5f, 0.25f))
6:             .Append(transform.DOScale(Vector3.one, 0.125f))
7:             .SetAutoKill(false)
8:             .Pause();
9:     }
10:
11:     public void Play() {
12:         _tween.Restart();
13:     }

```

Note that a tween that calls `SetAutoKill(false)` will leak if it is not explicitly destroyed. Call `Kill()` when it is no longer needed, or use the **SetLink** described below.

▼ List 12.3 Explicitly destroying a tween

```

1:     private void OnDestroy() {
2:         _tween.Kill();
3:     }

```

12.1.2 SetLink

Tweens that call `SetAutoKill(false)` or that are made to repeat indefinitely with `setLoops(-1)` will not be automatically destroyed, so you will need to manage their lifetime on your own. It is recommended that such a tween be associated with an associated `GameObject` at `SetLink(gameObject)` so that when the `GameObject` is Destroyed, the tween is also destroyed.

▼ List 12.4 Tethering a Tween to the Lifetime of a GameObject

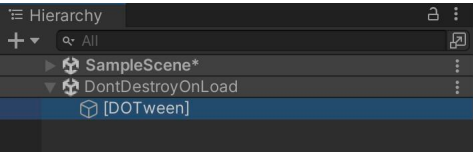
```

1:     private void Awake() {
2:         _tween = DOTween.Sequence()
3:             .Append(transform.DOScale(Vector3.one * 1.5f, 0.25f))
4:             .Append(transform.DOScale(Vector3.one, 0.125f))
5:             .SetAutoKill(false)
6:             .SetLink(gameObject)
7:             .Pause();
8:     }

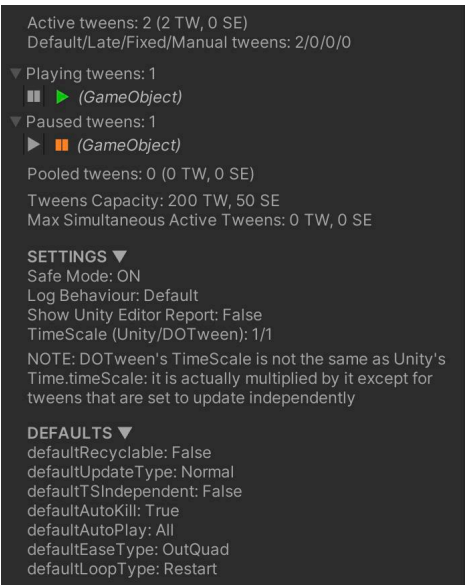
```

12.1.3 DOTween Inspector

During playback in the Unity Editor, a GameObject named **[DOTween]** You can check the state and settings of the DOTween from the Inspector by selecting the GameObject named



▲ Figure 12.1 [DOTween] GameObject



▲ Figure 12.2 DOTween Inspector

It is also useful to check for tween objects that continue to move even though their associated GameObjects have been discarded and for tween objects that are in a Pause state and leaking without being discarded.

12.2 UniRx

UniRx ^{*2} is a library implementing Reactive Extensions optimized for Unity. With a rich set of operators and helpers for Unity, event handling of complex conditions can be written in a concise manner.

12.2.1 Unsubscribe

UniRx allows you to subscribe (`Subscribe`) to the stream publisher `IObservable` to receive notifications of its messages.

When subscribing, instances of objects to receive notifications, callbacks to process messages, etc. are created. To avoid these instances remaining in memory beyond the lifetime of the `Subscribe` party, it is basically the `Subscribe` party's responsibility to unsubscribe when it no longer needs to receive notifications.

There are several ways to unsubscribe, but for performance considerations, it is better to explicitly `Dispose` retain the `IDisposable` return value of `Subscribe`.

```
1: public class Example : MonoBehaviour {
2:     private IDisposable _disposable;
3:
4:     private void Awake() {
5:         _disposable = Observable.EveryUpdate()
6:             .Subscribe(_ => {
7:                 // Processes to be executed every frame
8:             });
9:     }
10:
11:     private void OnDestroy() {
12:         _disposable.Dispose();
13:     }
14: }
```

If your class inherits from `MonoBehaviour`, you can also call `AddTo(this)` to automatically unsubscribe at the timing of your own `Destroy`. Although there is an overhead of calling `AddComponent` internally to monitor the `Destroy`, it is a good idea to use this method, which is simpler to write.

^{*2} <https://github.com/neuecc/UniRx>

```
1:     private void Awake() {
2:         Observable.EveryUpdate()
3:             .Subscribe(_ => {
4:                 // Processing to be executed every frame
5:             })
6:             .AddTo(this);
7:     }
```

12.3 UniTask

UniTask is a powerful library for high-performance asynchronous processing in Unity, featuring zero-allocation asynchronous processing with the value-based `UniTask` type. It can also control the execution timing according to Unity's `PlayerLoop`, thus completely replacing conventional coroutines.

12.3.1 UniTask v2

UniTask v2, a major upgrade of UniTask, was released in June 2020. UniTask v2 features significant performance improvements, such as zero-allocation of the entire async method, and added features such as asynchronous LINQ support and await support for external assets.^{*3}

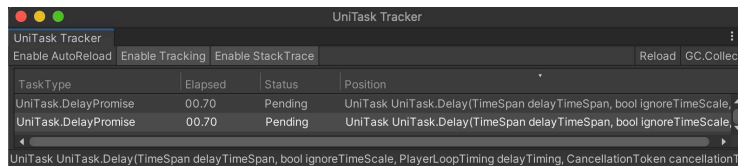
On the other hand, be careful when updating from UniTask v1, as it includes destructive changes, such as `UniTask.Delay(...)` and other tasks returned by `Factory` being invoked at invocation time, prohibiting multiple await to normal `UniTask` instances,^{*4} and so on. However, aggressive optimizations have further improved performance, so basically UniTask v2 is the way to go.

12.3.2 UniTask Tracker

UniTask Tracker can be used to visualize waiting UniTasks and the stack trace of their creation.

^{*3} <https://tech.cygames.co.jp/archives/3417/>

^{*4} `UniTask.Preserve` UniTask v2 can be converted to a UniTask that can be awaited multiple times by using



▲ Figure 12.3 UniTask Tracker

For example, suppose you have a `MonoBehaviour` whose `_hp` is decremented by 1 when it collides with something.

```

1: public class Example : MonoBehaviour {
2:     private int _hp = 10;
3:
4:     public UniTask WaitForDeadAsync() {
5:         return UniTask.WaitUntil(() => _hp <= 0);
6:     }
7:
8:     private void OnCollisionEnter(Collision collision) {
9:         _hp -= 1;
10:    }
11: }
```

If `_hp` of this `MonoBehaviour` is Destroyed before is fully depleted, `_hp` will not be depleted any further, so `UniTask`, the return value of `WaitForDeadAsync`, will lose the opportunity to complete, and will continue to wait.

It is recommended that you use this tool to check for `UniTask` leaking due to a misconfiguration of termination conditions.

Preventing Task Leaks

The reason why the example code leaks a task is that it does not take into account the case where the task itself is destroyed before the termination condition is met.

To do this, simply check to see if the task itself has been destroyed. Or, the `CancellationToken` obtained by `this.GetCancellationTokenOnDestroy()` to itself can be passed to `WaitForDeadAsync` so that the task is canceled when is Destroyed.

```
1:    // Pattern for checking whether the user is Destroyed or not
2:    public UniTask WaitForDeadAsync() {
3:        return UniTask.WaitUntil(() => this == null || _hp <= 0);
4:    }
5:
6:    // Pattern for passing a CancellationToken
7:    public UniTask WaitForDeadAsync(CancellationToken token) {
8:        return UniTask.WaitUntil(
9:            () => _hp <= 0,
10:           cancellationTokens: token);
11:    }
```

▼ List 12.9 Example of WaitForDeadAsync(CancellationToken) call

```
1:    Example example = ...
2:    var token = example.GetCancellationTokensOnDestroy();
3:    await example.WaitForDeadAsync(token);
```

At Destroy time, the former `UniTask` completes without incident, while the latter `OperationCanceledException` is thrown. Which behavior is preferable depends on the situation, and the appropriate implementation should be chosen.

CONCLUSION

This is the end of this document. We hope that through this book, those of you who are "not confident about performance tuning" have come to think, "I kind of get it, and I want to try it. As more people practice it in their projects, they will be able to deal with problems much faster, and the stability of their projects will increase.

You may also encounter complex events that cannot be solved with the information presented in this book. But even in such cases, what you will do will be the same. You will still need to profile, analyze the cause, and take some action.

From this point forward, please make full use of your own knowledge, experience, and imagination through practice. I hope you will enjoy performance tuning in this way. Thank you for reading to the end.

Introduction of the Authors

The following is a list of the authors involved in this book. Please note that the profiles of the authors and the sections they are responsible for are current at the time of writing.

Takuya Iida

Engineering Manager, SGE Core Technology Division, Grange Corporation. He is in charge of writing

Chapter 1 "Getting Started with Performance Tuning" and Chapter 3 "Profiling Tools". Currently involved in optimization across subsidiaries. I do various things in my work, and I am working hard everyday to improve development speed and quality.

Haruki Yano / Twitter: @harumak_11 / GitHub: Haruma-K

SGE Core Technology Division, CyberAgent, Inc. / Client-Side Engineer

He is in charge of writing "2.2 Rendering", "2.3 Data Representation" and other articles for Chapter 2 "Fundamentals". The main focus of my work is the development of common infrastructure to improve development efficiency. I develop and publish various OSS for Unity both in my work and personally. I am also operating the Unity blog LIGHT11.

Yusuke Ishiguro

CyberAgent, Inc. SGE Core Technology Division,

Responsible for part of Chapter 2 "Fundamentals" and writing for Chapter 5 "Tuning Practice - AssetBundle". He was assigned to the infrastructure development team of Ameba game (now QualArts) as a Unity engineer and engaged in the development of various infrastructures such as real-time infrastructure, chat infrastructure, Asset-Bundle management infrastructure "Octo", authentication and billing infrastructure. Currently, he is transferred to the SGE Core Technology Division, where he leads the overall infrastructure development and focuses on optimizing the development efficiency and quality of the entire Game Division.

Daiki Hakamata

Writes for

Chapter 9 "Tuning Practice - Script (Unity)" , SGE Core Technology Division, CyberAgent, Inc. Engaged in game development and operation at Grange Inc. and G-Crest Inc. Currently he belongs to SGE Core Technology Division and is developing the infrastructure.

Mitsutoshi Nakamura (NAKAMURO.) / Twitter: @megalo_23

Game creator and member of Applibot, Inc. He is in charge of writing the first half of "2.5 C# Basics" and Chapter 10 "Tuning Practice - Script (C#)". By writing this book, he plans to reduce the chances of being called in for help at the end of development, so that he can have more time to develop new games. His activities in game development range from optimization, direction, and music score creation to voice acting. On a personal basis, he is running an app at Famulite Lab.

Shunsuke Ohba / Twitter: @ohbashunsuke

Engineering Manager at Samzap Inc. and writer of

Chapter 4 "Tuning Practice - Asset" . Formerly an engineer and designer, Shunsuke Ohba joined CyberAgent, Inc. mid-career after creating interactive websites using Flash. After developing AmebaPig, switched to Unity engineer. He has participated in the launch of many games as an engineer leader, including mahjong, pinball, and real-time battle. On a personal basis, he provides information on Twitter and his blog "Shibuya Hottogisu Tsushin (<https://shibuya24.info>)".

Gaku Ishii

Server and client-side engineer at Samzap Inc. He writes for

Chapter 11 "Tuning Practice - Player Settings" and Chapter 12 "Tuning Practice - Third Party". After being assigned to Samzap Inc., he worked on the development of new game apps as a Unity engineer. After being involved in the release of several apps, he switched to server-side engineering. Currently working as a server-side engineer at Samzap and as a Unity engineer at SGE Core Technology Division, both on the server/client side.

Appendix Introduction of the Authors

Shunsuke Saito / Twitter: @shun_shun_mummy

He is a member of Colorful Palette Inc. / Client-side Engineer and writes some of the articles for

Chapter 10 "Tuning Practice - Script (C#)" . After being assigned to Colorful Palette Co., Ltd. he worked on design and implementation of client-side real-time communication and development around UI systems in the projects he was in charge of. He is also involved in tuning existing functions and developing tools for automatic generation of template codes.

Kazunori Tamura

He belongs to QualArts Corporation and writes for

Chapter 8 "Tuning Practice - UI" . At QualArts Corporation, he is engaged in game development and internal infrastructure development as a Unity engineer. He is mainly involved in the development of UI for the company's internal infrastructure. He is also interested in improving the efficiency of game development through AI, and is struggling to utilize AI within the game division.

Tomoya Yamaguchi / Twitter: @togucchi

Chapter 7 "Tuning Practice - Graphics" Client-side engineer at Colorful Palette Co. Engaged in the development of 3D rendering and live-related systems at Colorful Palette Co. Currently working on verification of new 3D-related technologies.

Yuichiro Mukai / Twitter: @yucchiy_

Client-side engineer at Applibot, Inc. He writes for

Chapter 6 "Tuning Practice - Physics" and some parts of Chapter 10 "Tuning Practice - Script (C#)".

Unity Performance Tuning Bible

Feb. 22, 2023 1st Edition

Author CyberAgent SGE Core Technology Team
Design CyberAgent Smartphone Games & Entertainment Division
Publisher CyberAgent, Inc.

(C) 2022 CyberAgent, Inc.



CyberAgent Smartphone Games & Entertainment

CyberAgent.