

进行测量的Unity编辑器不一定是你所建立的项目。建议为测量创建一个新的项目，因为它很轻便。

接下来，对于安卓系统，比iOS系统的步骤略多。

1. 从Build设置中把目标平台改为Android。
2. 将设备连接到电脑上，并启动开发构建应用程序。
3. 输入adb forward命令。(该命令的细节见下文)。
4. 从Unity Profiler中选择要连接的终端。
5. 开始记录

adb forward命令需要应用程序的软件包名称。例如，如果软件包名称是”jp.co.sample.app”，输入以下内容。

▼ 列表 3.2 adb forward命令

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

如果adb不被识别，请设置adb路径。省略了设置方法，因为网络上有很多解释设置方法的信息。

作为一个快速的故障排除，如果你不能连接，请检查以下内容

- 两种设备通用
 - 在已执行的应用程序的右下角是否有一个开发构建的符号？
- 适用于安卓系统
 - 终端上的USB调试功能是否启用？
 - 在adb forward命令中输入的软件包名称是否正确？
 - 当输入adb设备命令时，设备是否被正确识别。

作为补充，如果应用程序直接在Build And Run中执行，上述的adb forward命令将在内部执行。因此，测量时不需要命令输入。

自动连接程序

在构建配置中，有一个选项叫做“自动连接程序”（Autoconnect Profiler）。这个选项用于在应用程序启动时自动连接到编辑器的分析器。因此，它不是剖析的强制性设置。这同样适用于远程剖析：只有 WebGL 不能在没有这个选项的情况下进行剖析，但对于移动端来说，这不是一个非常有用的选项。

再进一步说，如果这个选项被启用，编辑器的 IP 地址在构建时被写入二进制文件，并在启动时尝试连接到该地址。例如，如果你在一个专用的构建机器上构建，这就没有必要，除非你在该机器上进行剖析。相反，它只会增加应用程序启动时自动连接超时的等待时间（约 8 秒）。

从脚本中，选项 `BuildOptions.ConnectWithProfiler`

要注意的是，名字是在名称中，很容易被误认为是强制性的。

3.1.2 CPU 使用率

CPU 使用情况显示如图 3.6 所示。

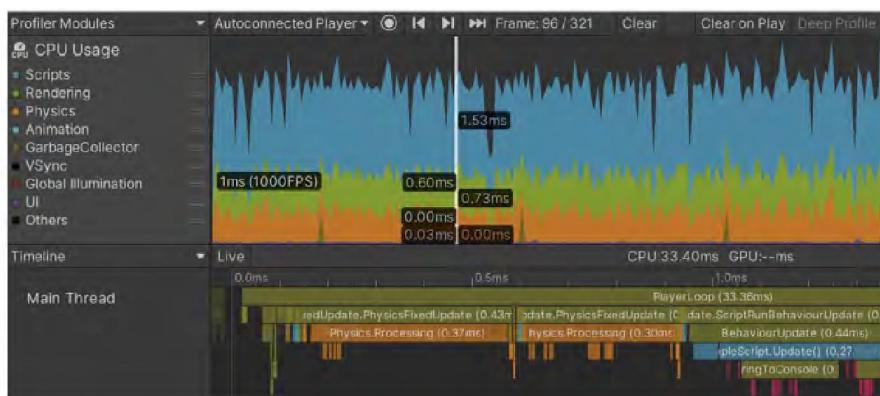


图 3.6 CPU 用量模块（时间轴显示）

有两种主要的方式来检查这个模块

- 层次结构（原始层次结构）。
- 时间轴。

首先，对“层次结构”视图所显示的内容和如何使用它进行了解释。

1. 层次结构视图

层次结构视图看起来像图3.7。

	Hierarchy	Live	Main Thread	CPU:33.46ms	GPU:--ms	Time ms	Self 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	
DebugUpdator.Update()	0.0%	0.0%	1	0 B	0.02	0.02	

图3.7 层次结构视图。

该视图的一个特点是，测量结果以列表形式排列，并可按标题中的项目进行排序。在进行调查时，可以通过从列表中打开感兴趣的项目来确定瓶颈。然而，显示的信息是在“选定的线程”上花费的时间的指示。例如，如果你使用作业系统或多线程渲染，另一个线程的处理时间就不包括在内。如果你想检查这一点，你可以通过选择一个线程来实现，如图3.8所示。

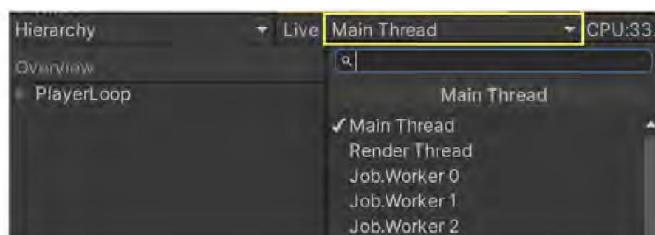


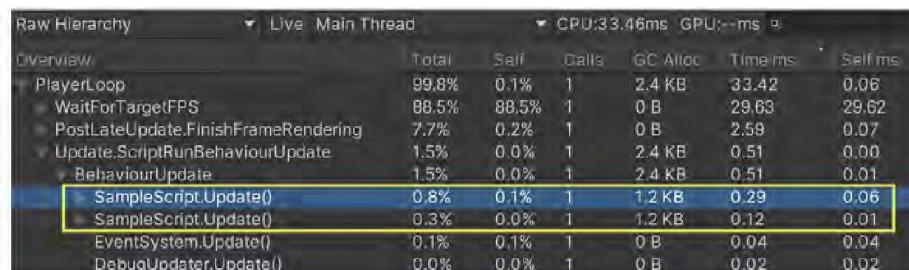
图3.8 线程选择

下一节将介绍标题项。

▼ 表3.1 层次结构标头信息

标题名称	描述。
概述。	样品名称。
共计	处理该功能所需的总时间(以%显示)
自己	这个函数本身的处理时间。不包括子功能的处理时间(以%显示)
呼叫。	在一帧内调用的次数。
GC分配。	本函数分配的脚本的堆内存。
时间ms	总的来说是毫秒。
自身毫。	自己在毫秒中。

Calls将几个函数调用合并为一个项目，作为一个视图更容易看到。然而，不清楚是所有的人都有相同的处理时间，还是只有一个人的处理时间更长。原始层次结构视图与层次结构视图不同的是，Calls总是固定在1。在图3.9中，Raw Hierarchy视图显示了对同一个函数的多次调用。



▲图3.9 原始层次结构视图。

总结到目前为止所讲的，层次结构视图用于以下目的

- 识别和优化减慢处理速度的瓶颈（Time ms, Self ms）。
- 理解并优化GC分配（GC Allocation）。

在执行这些任务时，建议在检查前对每个需要的项目进行降序排序。

在打开物品时，往往会有个很深的层次结构。在这种情况下，在Mac上按住Option键（或在Windows上按住Alt键），同时打开一个项目，就可以打开整个层次结构。反之，在按住键的同时关闭一个项目，就会关闭它下面的层次结构。

2. 时间轴视图

另一种检查时间线视图的方法如下。



图3.10 时间轴视图。

在时间线视图中，层次结构视图中的项目被可视化为方框，因此，在观察整个画面时，你可以直观地看到负载的位置，一目了然。而且，由于它是鼠标可及的，即使是很深的层次结构也可以拖动，以获得一个完整的概览。此外，在时间轴上，不需要在线程之间切换，它们都显示出来了。这使得我们很容易看到每个线程中正在做什么以及什么时候做。由于这些特点，它主要用于以下目的

- 我想了解一下处理负荷的总体情况。
- 想了解并调整每个线程的处理负荷

时间线不适合用于排序操作，以确定繁重的处理顺序，或检查分配的总量。因此，Hierarchy视图更适合于调整分配。

补充信息：关于采样器

有两种方法可以在每个功能的基础上衡量处理时间。
深度模型模式。另一种方法是直接嵌入到脚本中。

第3章 剖析工具

如果直接嵌入，请说明如下。

▼ 使用清单3. 3Begin/EndSample的方法

```
1: using UnityEngine.Profiling;
2: /* ... 缩略语... */
3: private void TestMethd()
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:     .
15:     TestMethod();
16:     Profiler.EndSample();
}
```

嵌入的样本在层次结构和时间线视图中都有显示。

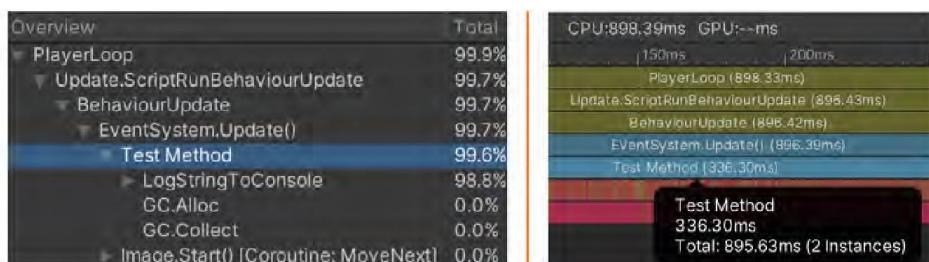


图3.11 采样器显示。

还有一个值得一提的特点。如果剖析代码不是Development Build，开销为零，因为调用者被禁用。在未来加工负荷可能增加的地区进行预建可能是一个好主意。

BeginSample方法是一个静态函数，可以很容易地使用，但也有一个具有类似功能的CustomSampler。这是从Unity 2017开始添加的，比BeginSample的测量开销更少，这意味着可以测量更准确的时间。

▼ 清单3.4 使用 CustomSampler的方法

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

不同的是，你需要提前创建一个实例；Custom- Sampler还允许你在测量后的脚本中获得测量时间。如果你需要更多的精度，或者你想根据处理时间发出警告，CustomSampler是一个不错的选择。

3.1.3 记忆

内存模块的显示如图3.12所示。

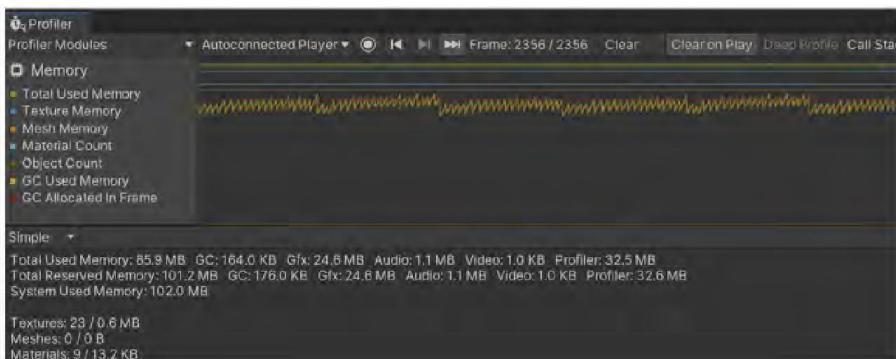


图3.12 内存模块。

有两种方法来检查这个模块

- 简单的观点
- 详细查看

首先，本节解释了显示的内容和如何使用简单视图。

1. 简单的观点

简单视图看起来像图3.13。

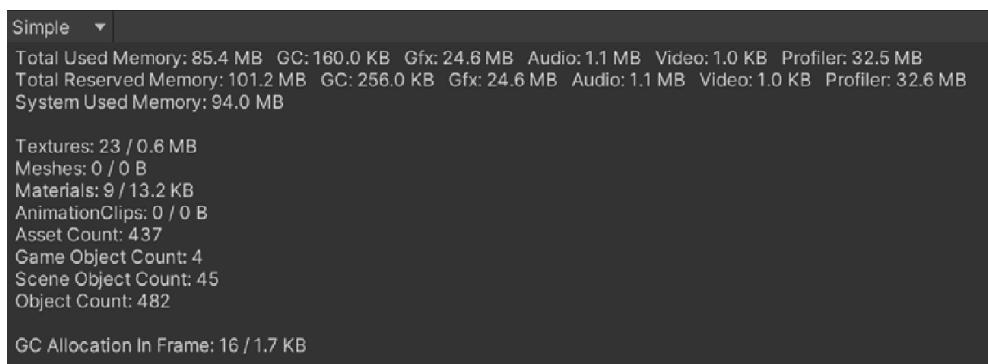


图3.13 简单视图。

本节介绍视图中列出的项目。

已用内存总量

Unity分配(使用中)的内存总量。

保留的总内存

目前Unity保留的内存总量；**在**操作系统端提前保留一定量的连续内存区域作为池，在需要时再分配。当池子的面积不够时，会再次向操作系统申请并扩大。

系统使用的内存

应用程序使用的总内存量。这个项目也衡量在总保留中没有衡量的项目（**缺口插件**）。然而，这仍然不能跟踪所有的内存分配。如果你想得到一个准确的情况，你需要使用一个本地兼容的剖析工具，如Xcode。

3.1 Unity Profiler

图3.13中 总使用内存右边列出的项目的含义如下。

▼ 表3.2 简单视图术语表

术语	描述。
ĀĀĀ	在堆区使用的内存量, 由于GC Alloc等因素而增加。
Gfx	纹理、着色器、网格等分配的内存数量。
音频	用于音频播放的内存量。
视频。	视频 用于播放的内存数量。
检察官。	用于剖析的内存量。

作为术语名称的补充说明, 从Unity 2019.2开始, "Mono"已被改名为 "GC", "FMOD"已被改名为 "Audio"。

图3.13还显示了使用的其他资产的数量和分配的内存量, 如下图所示。

- 纹理
- 网络。
- 材料
- 动画剪辑
- 音频剪辑

还有关于对象数量和GC分配的信息, 如下所示。

资产计数

装载的资产总数。

游戏对象数量

场景中存在的游戏对象的数量。

场景对象计数

场景中存在的组件、游戏对象等的总数量。

对象数量

由应用程序生成和加载的所有对象的总数。如果这个值在增加, 很可能是
一些对象在泄漏。

框架中的GC分配

在一个框架内发生分配的次数和总量。

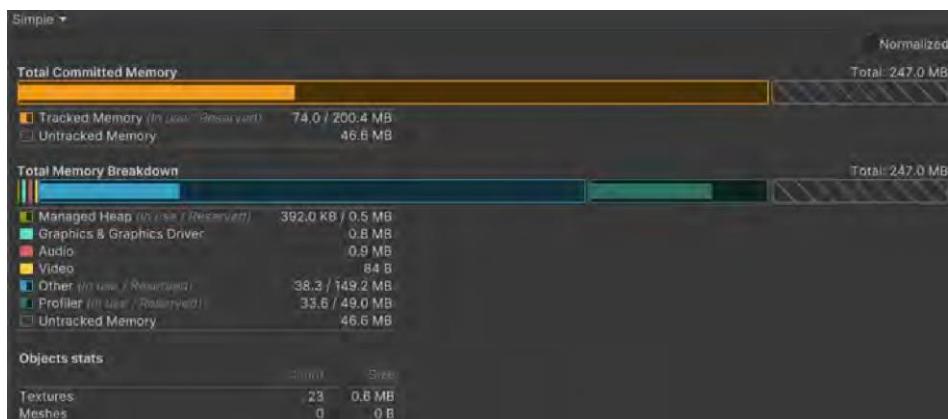
3.1 Unity Profiler

最后，从这些信息中总结出简单视图的用例。

- 了解并监控堆积面积和保留的扩展时间
- 检查各种资产和物体的泄漏情况。

- GC分配监测

Unity 2021及以后版本的“简单”视图的用户界面有了很大的改进，使其更容易看到显示的项目。内容本身没有大的变化，所以你仍然可以使用这里介绍的知识。然而，请注意，有些名字已经被改变。例如，GC已被重新命名为管理堆。



▲ 图3.14 从2021年起的简单视图。

2. 详细查看

Detailed视图看起来像图3.15。

Name	Memory	Ref count	Referenced By
Other (81)	44.8 MB		
Assets (420)	1.5 MB		
AudioManager (1)	1.1 MB		
Shader (7)	233.3 KB		
Hidden/Internal-GUIRoundedRectWithColorPerBorder	45.2 KB	2	
Skybox/Procedural	44.8 KB	1	
Hidden/Internal-GUIRoundedRect	40.5 KB	2	
Sprites/Default	32.2 KB	3	SplashScreen-Foreground(Material) GraphicsSettings(GraphicsSettings) Sprites-Default(Material)
Hidden/Internal-GUITextureClip	31.9 KB	3	
Hidden/Internal/GUITexture	24.0 KB	3	
Hidden/BillCopy	14.6 KB	2	
MonoScript (396)	135.9 KB		

图3.15 详细视图

按“取样”按钮，可以把这个显示的结果作为当时的记忆快照；与“简单”视图不同，这不是实时更新，所以如果你想刷新显示，你需要再次取样。

在图3.15的右侧，你会看到“引用者”这一项。这显示了引用当前所选对象的对象。如果有资产泄漏，关于对象的参考信息可能有助于解决问题。这个显示只有在“收集对象参考”被启用时才会显示。启用该功能会增加取样时的处理时间，但一般建议不启用。

你可能会看到ManagedStaticReferences()这个符号来表示Referenced By。这意味着它被某个静态对象所引用。如果你熟悉这个项目，这些信息可能足以让你有所了解。如果没有，建议使用3.5堆栈浏览器。

详细视图中的标题项目在此不作解释，因为它们的含义与看上去的含义相同。操作方法与“3.1.2 CPU使用情况”中的“1. 分层视图”相同，对每个标题有排序功能，对项目有分层视图。本节对“名称”项目中显示的顶部节点进行解释。

▼ 表3.3 详细的顶级节点

名字。	描述。
资产	场景中不包括已加载的资产。
未获救。	由代码在运行时生成的资产。 一个由代码生成的对象，例如，new Materiala()。
场景记忆	包括在加载场景中的资产。
其他。	对上述对象以外的、被Unity在不同系统中使用的对象的赋值。

你可能不熟悉顶部节点中其他项下所列的项目。以下是一些值得了解的项目。

系统.可执行程序和文件

表示用于二进制文件、DLLs等的分配数量。它可能不适用于某些平台或终端，在这种情况下，它被当作0B处理。项目的内存负荷并不像所述值那么大，因为它可能与其他使用共同框架的应用程序共享。与其急于减少这一项，还不如改善资产。最有效的方法是减少DLLs和不必要的Scripts。最简单的方法是改变剥离水平。然而，在运行时有遗漏类型和方法的风险，所以要仔细调试。

串行化文件

表示元信息，如AssetBundle中的对象表或作为类型信息的类型树。这可以通过AssetBundle.Unload(true or false)释放。最有效的方法是使用Unload(false)在资产加载后只释放这些元信息，但要注意，如果释放时机和资源引用管理做得不仔细，资源会被重复加载，容易发生内存泄漏。

PersistentManager.Remapper.

它管理内存中的对象和磁盘上的对象之间的关系；Remapper使用一个内存池，当它用完时，内存池会以翻倍的方式扩展，不会减少。必须注意不要过度膨胀。具体来说，如果加载了大量的AssetBundles，映射区域将不够用，会被扩大。因此，卸载不必要的AssetBundles是一个好主意，以减少同时加载的文件数量。如果一个资产包包含大量不需要的资产，将其拆分也是一个好主意。

最后，我们总结了到目前为止所介绍的使用详细视图的案例。

- 详细了解和调整特定时序下的内存
 - 没有不需要的或意外的资产
- 调查内存泄漏。

3.2 档案分析器

Profile Analyzer是对Profiler的CPU Usage获得的数据进行更详细分析的工具：Unity Profiler只允许你查看每一帧的数据，Profile Analyzer允许你根据指定帧的间隔获得平均数、中位数、最小和最大值。值、中位数、最小值和最大值，基于指定的框架区间。这可以让你适当地处理每一帧不同的数据，这样你就可以在优化时更清楚地显示改进的效果。它也是一个非常有用的工具，用于比较和可视化优化的结果，因为它有一个测量数据之间的比较功能，而CPU Usage无法做到。

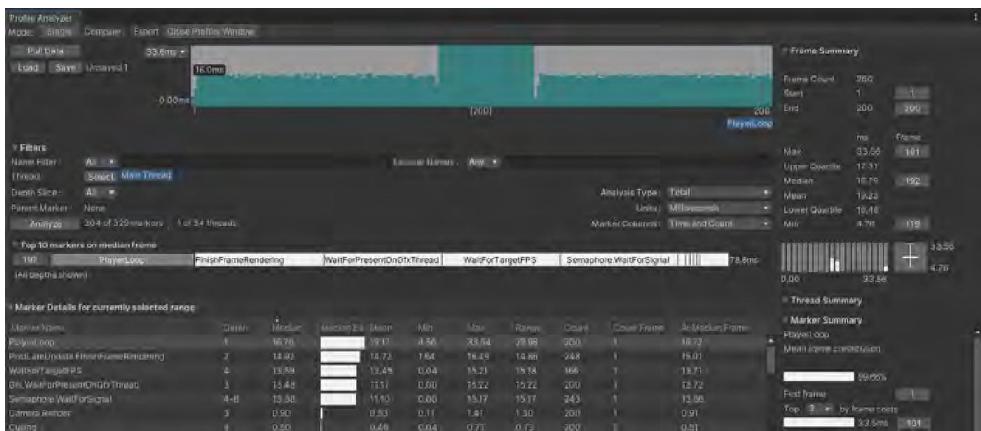


图3.16 Profile分析仪。

3.2.1 介绍的方法

该工具可以从Unity官方支持的Package Manager中安装，将Packages改为Unity Registry，在搜索框中输入’Profile’。安装后可以通过”窗口->分析->模板分析器”来启动该工具。



图3.17 从PackageManager安装

3.2.2 如何操作

启动后，Profile Analyzer立即看起来像图3.18。

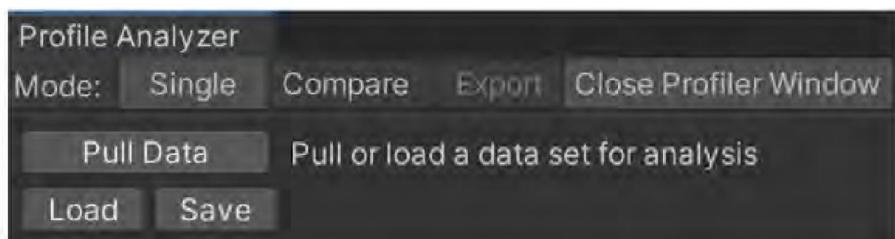


图3.18 开机后立即投入使用

有两种功能模式：“单一”和“比较”；单一模式用于分析单一测量，而比较模式用于比较两个测量数据。

“拉数据”允许你分析用Unity Profiler测量的数据并显示结果。提前用Unity Profiler测量。

“保存”和“加载”允许你保存和加载Profile Analyzer所分析的数据。当然，如果你只想保留Unity Profiler中的数据，那就没有问题。在这种情况下，你必须在Unity Profiler中加载数据，并在Profile Analyzer中每次做Pull Data。如果该程序很麻烦，最好将数据保存为专用数据。

3.2.3 分析结果（单模式）

分析结果屏幕的结构如下。标记这个词出现在这里，指的是进程的名称(方法名称)

◦

- 分析部分设置屏幕。
- 显示项目过滤器输入屏幕。
- 标记的中值 前10名
- 标志物分析结果。
- 框架摘要
- 主题摘要
- 所选标记的摘要

让我们来看看每一个显示屏幕。

1. 分析部分设置屏幕。

每一帧的处理时间被显示出来，所有的帧都被初步选中。如图3.19所示，帧间隔可以通过拖动来改变，必要时可以调整。



图3.19 框架部分的指定

2. 过滤器输入屏幕

过滤输入屏幕允许对分析结果进行过滤。

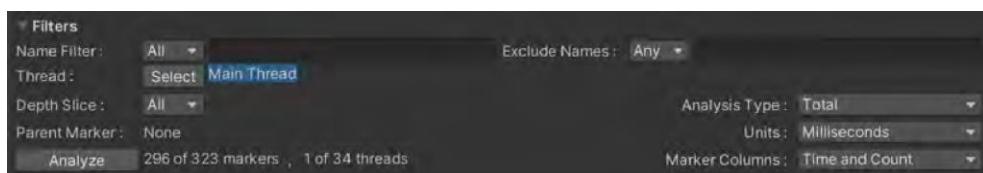


图3.20 过滤器输入屏幕。

3.2 序列分析器

每项内容如下。

▼ 表3.4 过滤器条目。

项目名称	描述。
名称过滤器	通过你要搜索的过程的名称进行过滤。
排除过滤器	通过你想从搜索中排除的过程的名称进行过滤。
课题。	选择的线程会显示在分析结果中。 如果你需要, 在其他线程上添加信息。
深度切片	在CPU使用率中介绍的层次结构显示的数量。 例如, 如果深度是3, 就会显示第三个层次。
分析类型	CPU使用情况 允许在总数和自身之间切换, 如在标题项目中发现的那样。
单位。	时间显示可以改成毫秒或微秒。
标记栏。	可以改变分析结果的标题显示。

当深度切片设置为全部时, 就会显示名为PlayerLoop的顶层节点, 或者显示同一进程的不同层, 这样就很难看清。在这种情况下, 建议将深度固定为2~3, 并设置为显示渲染、动画和物理等子系统。

3. 标记的中值 前10名

这个屏幕按照中值对每个标记的处理时间进行排序, 只显示前10个标记。

下表显示。你可以一目了然地看到前十个标记中的每一个占用了多少处理时间。



▲ 图3.21 标志物的前10个中位值

4. 标志物分析结果。

每个标志物的分析结果都会显示出来; 你不妨根据标志物名称下所列的治疗名称以及中位数和平均值来分析要改进的治疗。如果你把鼠标指针移到一个标题项目上, 就

第3章 剖析工具

会显示这个项目的描述，这样你就可以在不确定它的内容时参考它。

3.2 序列分析器

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

▲ 图3.22 每个处理的分析结果

平均数和中位数

平均值是将所有数值相加后除以数据数得到的数值。与此相反。

中位数是排序后的数据中间的数值。如果有偶数个数据，则从中位数前后的数据中取平均值。

平均数有受到数值相差极大的数据影响的倾向。如果经常出现尖峰或采样数量不足，参考中位数可能更好。

图3.23显示了一个中位数和平均值之间存在巨大差异的例子。



▲ 图3.23 中位数和均值

在了解这两个数值的特点后，分析数据。

5. 框架摘要

该屏幕显示测量数据的帧统计。

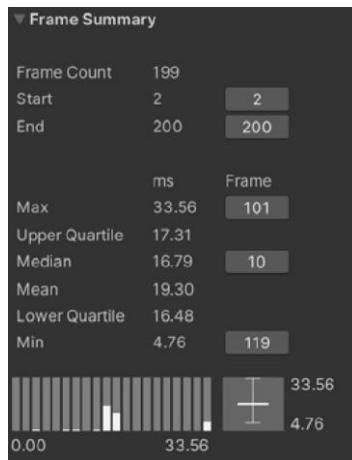


图3.24 框架摘要屏幕。

关于被分析的帧的间隔和数值的变化程度的信息是用boxplots和柱状图显示的。箱形图需要对四分位数的理解。四分位数是定义值，数据排序如表3.5所示。

▼ 表3.5 四分位数

名称。	描述。
最小值 (Min)	闵行区。
下四分位数	价值在最低价值的25%处
中位数 (中位数)	价值在最低值的50%处
上四分位数	从最低值开始的75%位置的价值
最大值 (Max)	最大值。

25%和75%之间的区间被框住了，这就叫盒须图。

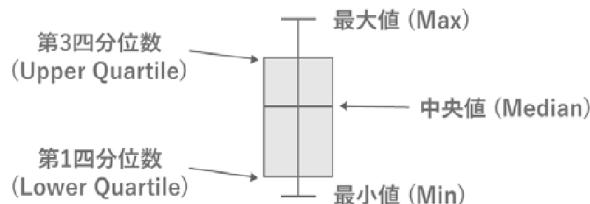


图3.25 箱形和胡须图

直方图在横轴上显示处理时间，在纵轴上显示数据数量，这对于查看数据分布也很有用。在帧摘要中，将光标悬停在帧上就可以查看间隔和帧数。

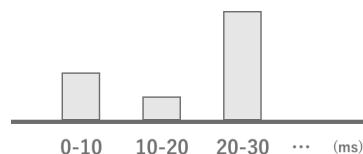


图3.26 直方图。

最好是了解如何看待这些图表，然后分析其特点。

6. 主题摘要

该屏幕显示所选线程的统计数据。可以查看每个线程的箱形和胡须图。

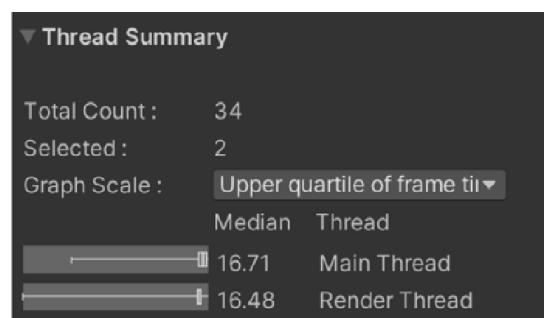


图3.27 框架摘要屏幕。

7. 所选标记的摘要

4. 在‘4. 标记分析结果’屏幕中选择的标记的摘要。当前所选标记的处理时间以盒须图或直方图的形式显示。

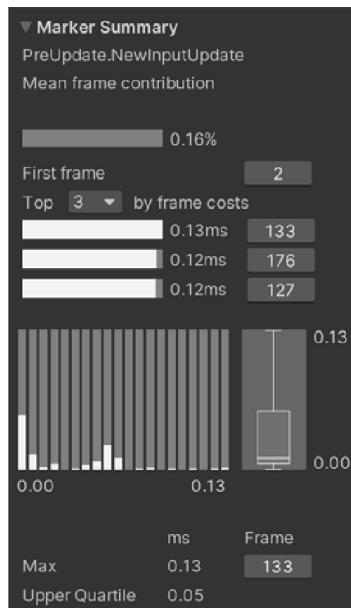


图3.28 选择中的标志物摘要

3.2.4 分析结果（比较模式）。

在这种模式下，两组数据可以进行比较。可以为每个上层和下层数据设置要分析的间隔。

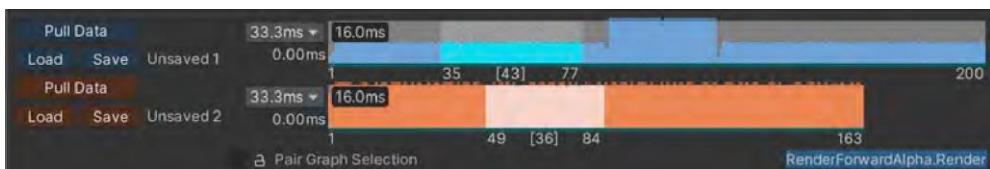


图3.29 设置比较数据

3.3 框架调试器

屏幕的使用与单人模式基本相同，但“左”和“右”字会出现在不同的屏幕上，如图3.30所示。

Marker Name	Left Median	Right Median	Diff	Abs Diff	Col
Gfx.WaitForGfxCommandsFromMainThread	0.66	31.85	31.19	31.19	78
WaitForTargetFPS	13.90	32.44	18.55	18.55	43
Semaphore.WaitForSignal	14.84	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

▲ 图3.30 标记的比较

这表明哪个数据是哪个，并与图3.29中显示的颜色一致；左边是来自顶部的数据，右边是来自底部的数据。这种模式将使我们更容易分析好的和坏的调整结果。

3.3 框架调试器

帧调试器是一个工具，它允许你分析当前显示的屏幕是如何被渲染的。该工具默认安装在编辑器中，可以通过选择“窗口->分析->框架调试器”打开。

它可以在编辑器或实际设备上使用。当在真正的机器上使用时，如Unity Profiler需要一个用“开发构建”构建的二进制文件。启动应用程序，选择设备连接点，并按“启用”来显示绘图说明。

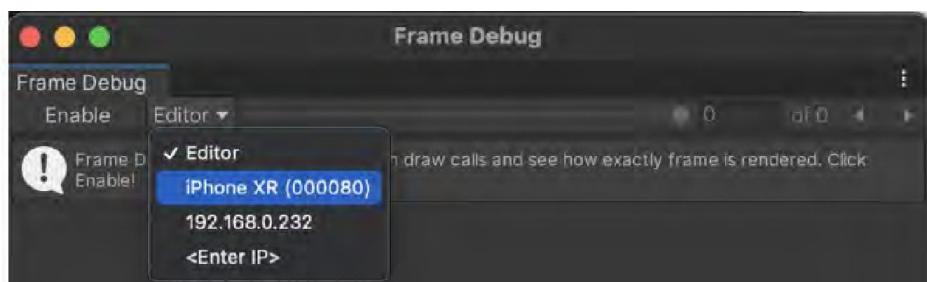


图3.31 FrameDebugger连接屏幕。

3.3.1 分析屏幕

按“启用”，你将看到以下屏幕。

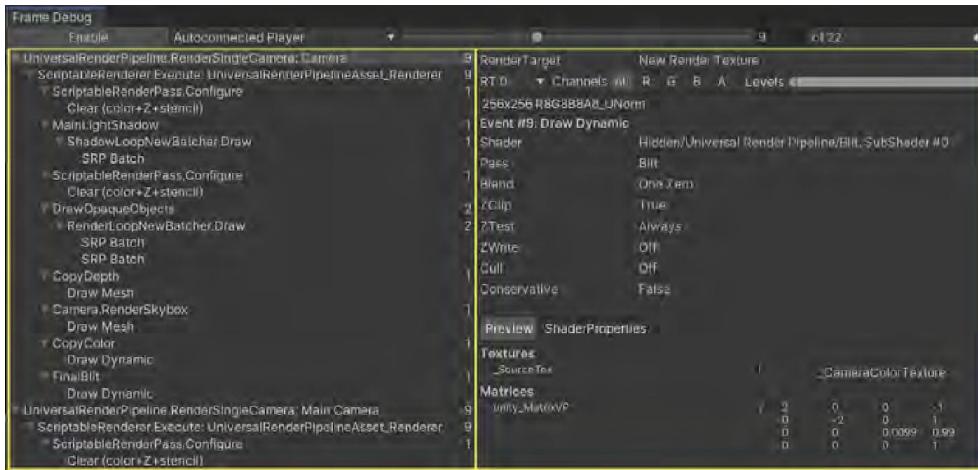


图3.32 FrameDebugger捕获。

左边的框架显示一个项目为一个绘图指令，指令从上往下依次发出。右边的框架显示关于绘图指令的详细信息。你可以看到哪些着色器被处理，以及哪些属性。在看这个屏幕的时候，要注意进行以下的分析。

- 是否有任何不必要的命令？
- 图纸批改没有适当的效果或无法总结
- 绘图目标的分辨率是否过高？
- 你是否使用了一个非预期的着色器？

3.3.2 详细屏幕

上一节介绍的图 3.32 中的右边框详细解释了。

控制面板

3.3 框架调试器

首先, 让我们来谈谈上部的控制面板。



▲图3.33 上部操作面板

当存在多个渲染目标时，RT0可以被改变。通道允许你改变是显示所有的RGBA还是只显示其中一个通道。等级是一个滑块，允许你调整绘图结果的亮度。这是非常有用的，例如，当渲染是黑暗的，如在环境或间接照明下，调整亮度，使其更容易看到。

绘图纲要

在这个区域，你可以找到关于图纸目的地的分辨率和格式的信息。很明显，如果有一个分辨率更高的渲染器，你会立即注意到。你还可以看到其他信息，如正在使用的着色器的名称、Cull等通行证设置以及正在使用的关键词。底部的句子“为什么这样~”描述了为什么这幅画不能被批改。在图3.34的情况下，它指出批处理是不可能的，因为第一个绘图调用被选中。由于原因描述得如此详细，如果你想设计一个批处理过程，你可以依靠这些信息来调整批处理。

```
824x1210 B10G11R11_UFloatPack32
Event #13: SRP Batch
Draw Calls          3
Shader              Universal Render Pipeline/Lit, SubShader #0
Pass                ForwardLit (UniversalForward)
Keywords            _MAIN_LIGHT_SHADOWS
Blend               One Zero
ZClip               True
ZTest               LessEqual
ZWrite              On
Cull                Back
Conservative       False

Why this draw call can't be batched with the previous one
SRP: First call from ScriptableRenderLoopJob
```

▲图3.34 中间部分的绘图概述。

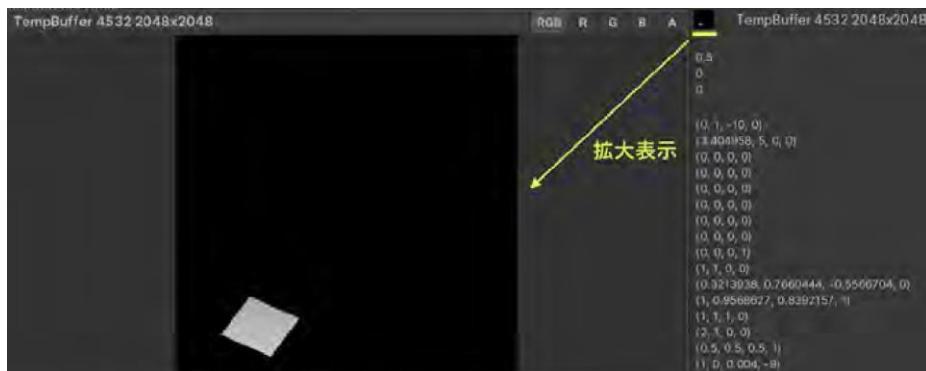
关于着色器属性的更多信息。

这个区域包含正在绘制的着色器的属性信息。这对调试是很有用的。

Preview			ShaderProperties		
Textures					
unity_SpecCube0	f				
_BaseMap	f		UnityWhite		
_MainLightShadowmapTexture	f		TempBuffer 1	2048x2048	
FLOATS					
_Smoothness	f	0.5			
_Metallic	f	0			
_Surface	f	0			
Vectors					
_WorldSpaceCameraPos	vf	(0, 1, -10, 0)			
unity_OrthoParams	v	(2.310268, 5, 0, 0)			

▲ 图3.35 关于下层着色器属性的详细信息。

有时你可能想详细检查属性信息中显示的Texture2D处于什么状态。要做到这一点，在Mac上按住Command键（在Windows上按住Control键），然后点击图像进行放大。



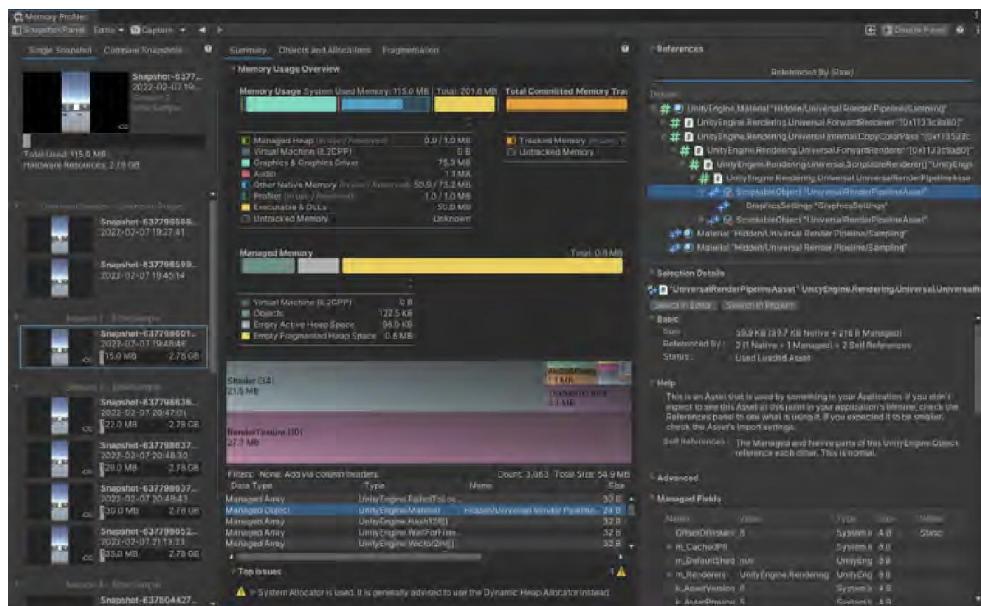
▲ 图3.36 放大Texture2D预览

3.4 储存器

Memory Profiler是Unity作为预览包提供的官方工具,与Unity Profiler的Memory模块相比,它有以下主要优势

- 捕获的数据与屏幕截图一起存储在本地。
- 可视化和易于理解的每个类别的内存占用情况
- 数据可以进行比较。

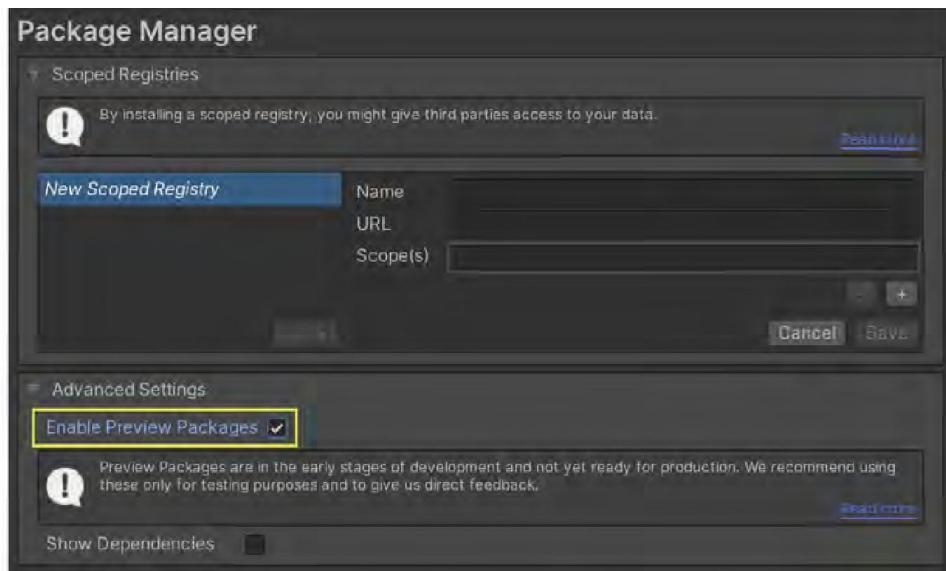
从0.4版本开始,Memory Profiler的用户界面发生了重大变化。本书使用0.5版,这是写作时的最新版本。对于0.4及以上版本,需要Unity 2020.3.12f1或更高版本来使用所有功能。此外,v0.4和v0.5乍一看是一样的,但v0.5在功能上有很大更新。特别是,现在的对象引用非常容易遵循,所以基本上推荐使用v0.5或更高版本。



▲ 图3.37 内存管理器。

3.4.1 介绍的方法

在Unity 2020中, 预览版本的软件包必须在 “项目设置->软件包管理器” 中启用 ”启用预览软件包”。



▲图3.38 激活预览包

然后从Unity注册表的软件包中安装Memory Profiler。安装后, 该工具可以通过 ”窗口->分析->内存探测器” 启动。



图3.39 从PackageManager安装

另外,从Unity 2021开始,添加软件包的方式也发生了变化。要添加一个软件包,你需要按”按名称添加软件包”,并输入”com.unity.memoryprofiler”。



▲ 图3.40 2021年后如何添加

3.4.2 如何操作

记忆程序由四个主要元素组成。

- 工具栏
- 快照小组
- 测量结果
- 详见面板

将对这些领域中的每一项进行解释。

1. 工具栏

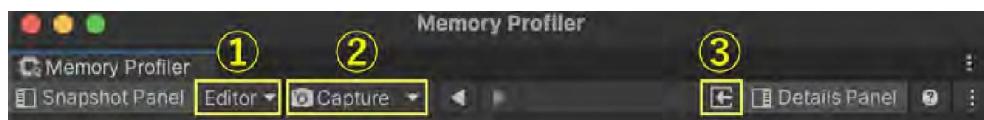


图3.41 工具栏区域。

图3.41显示了头的捕捉。按钮(1)允许你选择测量目标。按键(2)在按下时测量记忆。可选的是，测量目标可以设置为只针对本地对象，或者禁用屏幕截图。使用基本的默认设置不会有问题。按钮③按下后可以读取测量数据。通过按下“快照面板”或“细节面板”按钮，你可以显示或隐藏左右两边的信息面板。你也可以点击“？”来打开官方文档。

关于测量，有一个注意事项：测量所需的内存是新分配的，此后不再释放。关于测量要注意的一点是，测量所需的内存是新分配的，不会再被释放。然而，它不会无限制地增加，经过几次测量后最终会稳定下来。在测量时分配的内存量将取决于项目的复杂性。请注意，如果你不知道这个假设，当你看到内存使用量膨胀时，你可能会错误地认为有泄漏。

2. 快照小组

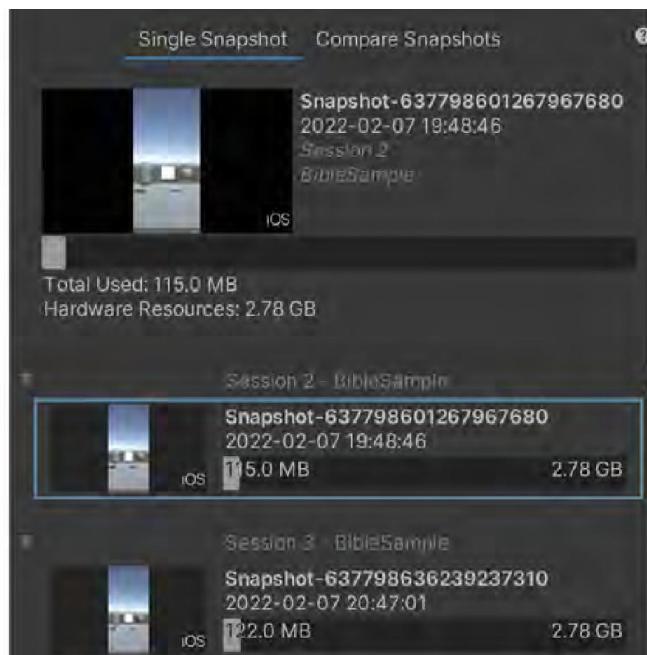


图3.42 快照面板（单件）

快照面板显示测量数据，并允许你选择查看哪些数据。这些数据是按会话分组的，从应用开始到会话结束。你还可以通过右键点击删除或重命名测量的数据。

在顶部是'单一快照'和'比较快照'.比较快照。

按一下会改变显示方式，变成一个用于比较测量数据的用户界面。

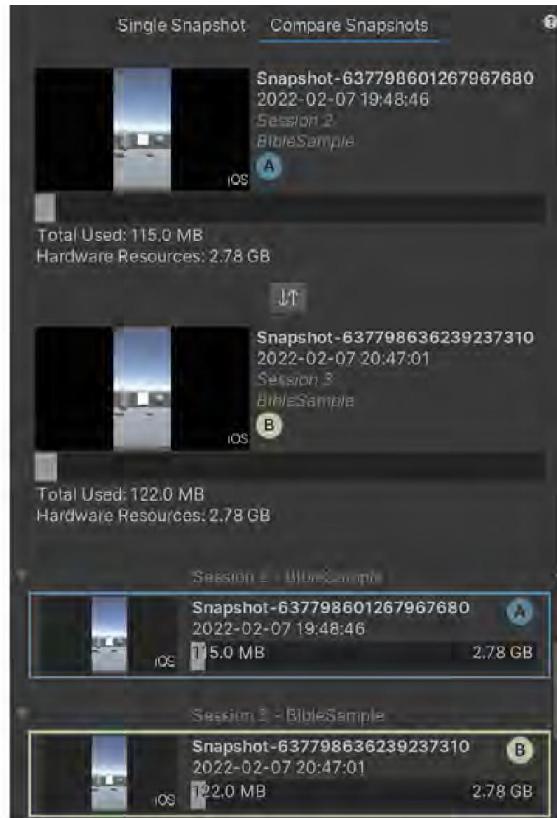


图3.43 快照面板 (Comapre)。

A'是单一快照中选择的数据，'B'是组合快照中选择的数据。你可以通过按交换按钮在"A"和"B"之间进行切换，因此你可以在它们之间进行切换，而不必一直回到"单张快照"屏幕。

3. 测量结果

测量结果显示在三个标签中：摘要、对象和分配以及碎片化。摘要屏幕的上半部分被称为“内存使用概况”，显示了当前内存的概况。点击一个项目会在细节面板上显示解释，所以检查你不明白的项目是一个好主意。

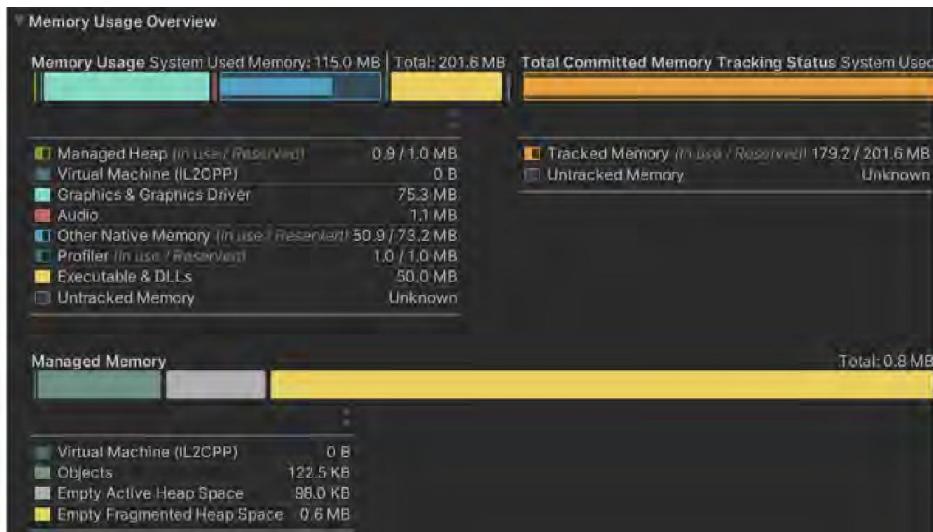


图3.44 内存使用概况

屏幕的下一部分被称为“树状图”，以图形方式显示每一类对象的内存使用情况。选择每个类别，你就可以看到该类别中的对象。在图3.45中，Texture2D类别被选中。

3.4 存储器程序



图3.45 树木图。

然后，屏幕的下半部分被称为树状图表。在这里，对象的列表以表格形式排列。通过按下树状图表的标题，可以对显示的项目进行分组、排序和过滤。



▲ 图3.46 头部操作。

特别是，对类型进行分组使其更容易分析，应积极主动地使用。

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

图3.47 按类型分组。

第3章 剖析工具

另外,如果在树状图中选择了一个类别,就会自动设置一个过滤器,只显示该类别的对象。

The screenshot shows a table titled 'Count: 34 Total Size: 21.5 MB' with three filters at the top: 'Type' (Is Shader), 'Data Type' (Is Native Object), and 'Sort' (Size). The columns are Data Type, Type, Name, Size, Referenced By, and Value. The data includes various shader components like Hidden/Universal Render Pipeline... and Universal Render Pipeline/Lit.

Data Type	Type	Name	Size	Referenced By	Value
Native Obj...	Shader	Hidden/Universal Render Pipeline...	19.1 MB	4	0x000000010618ca10
Native Obj...	Shader	Hidden/Universal Render Pipeline...	417.1 KB	3	0x000000010618e610
Native Obj...	Shader	Hidden/Universal Render Pipeline...	217.8 KB	4	0x000000010617d210
Native Obj...	Shader	Hidden/Universal Render Pipeline...	217.2 KB	4	0x000000010613b810
Native Obj...	Shader	Hidden/Universal Render Pipeline...	211.3 KB	4	0x0000000106150210
Native Obj...	Shader	Hidden/Universal Render Pipeline...	195.4 KB	4	0x000000010616a810
Native Obj...	Shader	Universal Render Pipeline/Lit	181.4 KB	1	0x0000000101832210

图3.48 自动过滤器设置。

最后,描述了使用比较快照时的用户界面变化:内存使用概述显示了每个对象的差异。

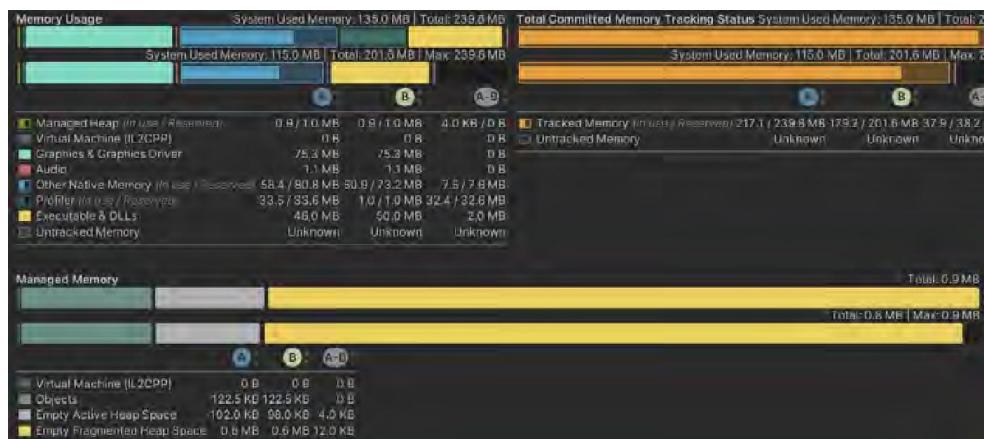


图3.49 比较快照的内存使用概况。

树状图表还为标题添加了一个Diff条目,它有以下类型

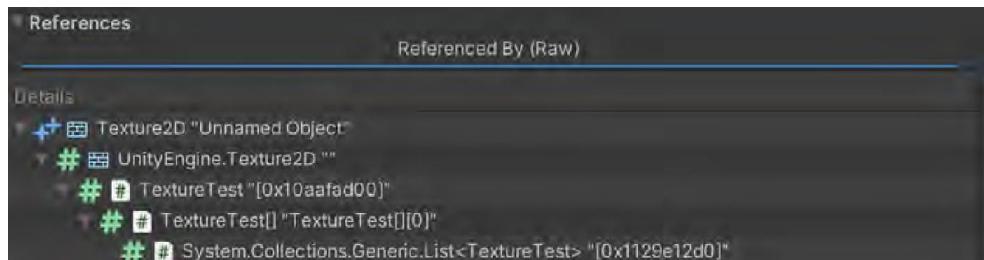
▼ 表3.6 树状图表（比较）

扩散。	描述。
一样。	A, B 相同的物体
不在A区(已删除)。	A中的物体, 但B中没有
不在B区(新)	物体不在A处而在B处

你可以通过查看这些信息来检查内存是否在增加或减少。

4. 详见面板

当你想跟踪所选对象的参照关系时, 就会用到这个面板。通过检查这个引用者, 你将能够确定持续引用抓取的原因。



▲图3.50 参考文献。

底部的“选择细节”部分包含了关于该对象的详细信息。除其他事项外, 帮助部分还包含关于如何发布的建议。如果你不确定该怎么做, 不妨阅读一下。

第3章 剖析工具

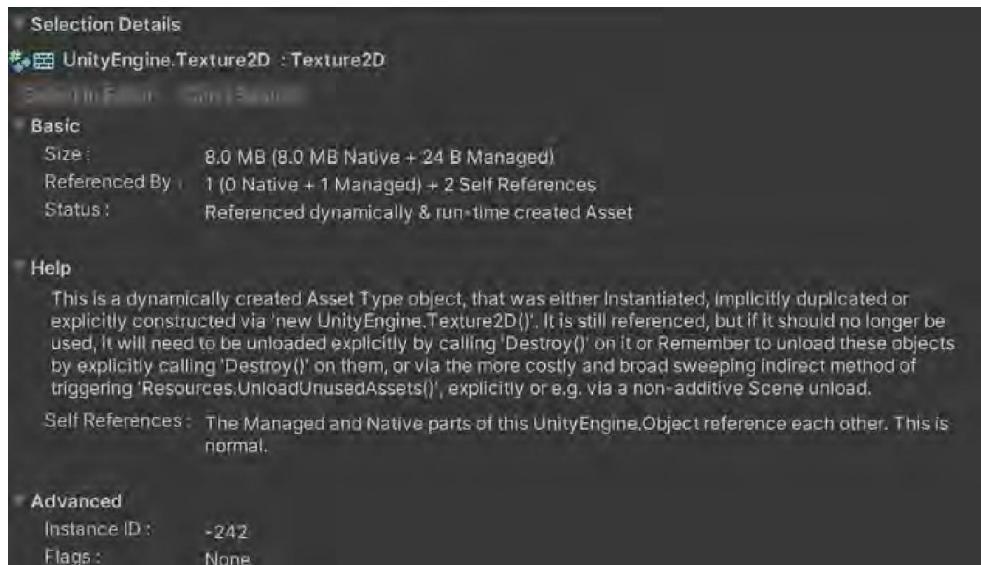


图3.51 选择细节。

补充信息：除摘要外的测量结果

对象和分配 "与 "摘要 "不同的是，更详细的细节，如分配，可以以表格形式查看。

Select Table View:	All Objects ▾
Filters:	None, Add v
Data Type	Raw Data
Managed Array	All Native Allocations (27,903)
Managed Object	All Managed Objects (2,206)
Managed Object	All Native Objects (1,003)
Managed Object	✓ All Objects (3,209)
Managed Object	UnityEngine.Material
Managed Array	UnityEngine.SliderState[]
Managed Object	UnityEngine.Texture2D
Managed Object	UnityEngine.Material
Managed Array	UnityEngine.Animator[]
Managed Array	UnityEngine.Animations...
	Root Reference (1,087)
	Native Allocation (27,903)
	Native Memory Label (161)
	Native Memory Regions (89)
	All Memory Regions (92)
	Native Object (1,003)
	Native Type (86)
	Native Type Base (236)
	Native Connection (1,209)
	Managed Type (9,111)

▲图3.52 指定一个表视图

3.4 存储器程序

“Fragmentation”将虚拟内存的情况可视化，并可用于调查碎片化。它可能很难使用，因为有很多非直观的信息，如内存地址。

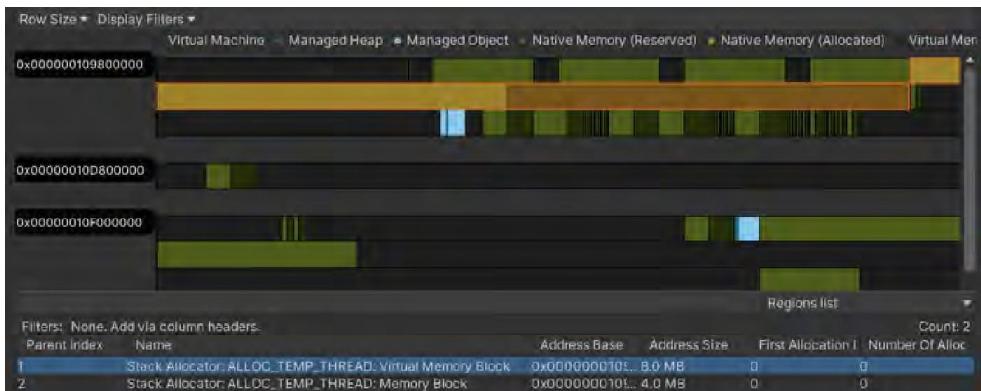


图3.53 分裂。

Memory Profiler v0.6增加了一个名为“Memory Breakdowns”的新功能，它需要Unity 2022.1或更高版本，但允许你在列表视图中查看TreeMaps和Unity Subsystems等对象信息。现在还可以检查可能的重复对象。

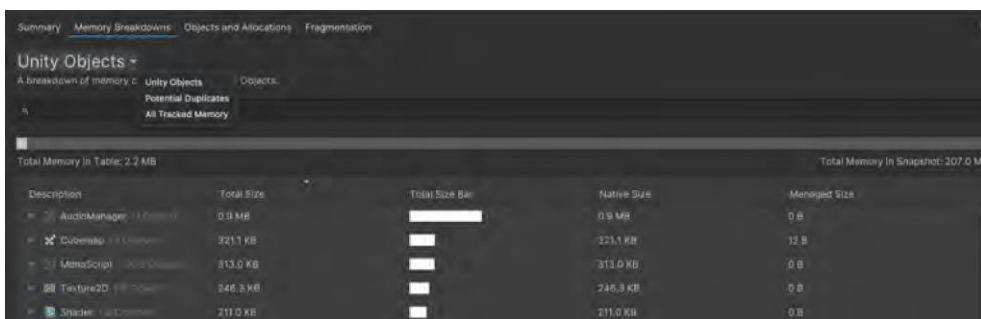


图3.54 内存分解。

3.5 堆栈资源管理器。

Heap Explorer是一个开源工具，来自私人开发者Peter^{77*1}。与Memory Profiler一样，这也是一个经常被用来调查内存的工具。在0.4之前的版本中，由于参考文献没有以列表的形式显示，所以Memory Profiler的追踪工作非常费力，虽然这在0.5之后得到了改进，但有些人可能使用的Unity版本不支持。在这种情况下，它作为一种替代工具仍有很大的价值，所以我们想在本文中介绍它。

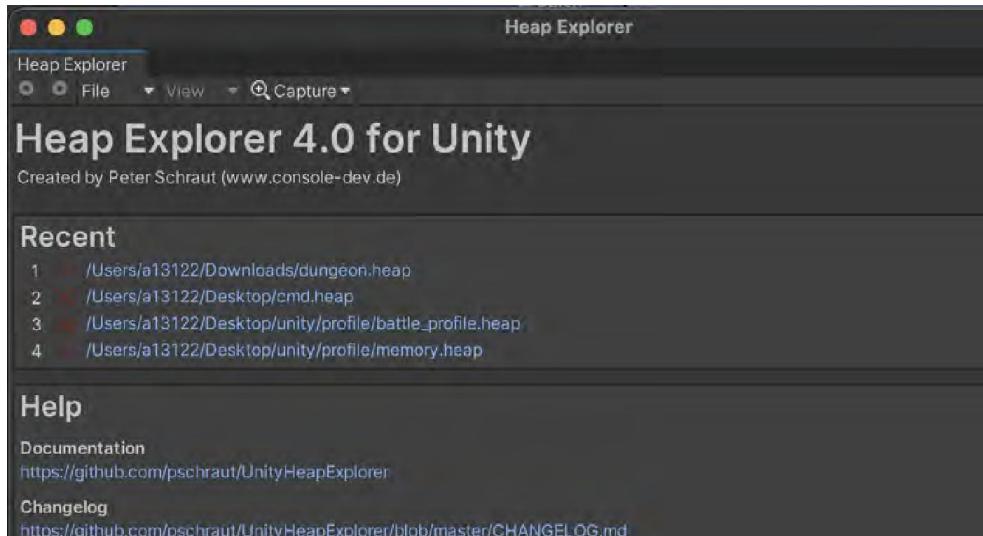


图3.55 堆浏览器。

3.5.1 介绍的方法

该工具复制GitHub存储库^{*2}中列出的软件包URL，然后从软件包管理器中的 "Add Package from Git url" 添加该软件包。安装后该工具可以在 "窗口->分析->内存探测器" 下启动。

3.4 存储器程序

*¹ <https://github.com/pschraut>

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

3.5.2 如何操作

Heap Explorer的工具栏看起来像这样

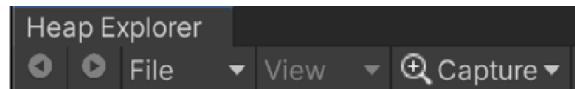


图3.56 堆浏览器工具栏。

左边和右边的箭头

允许你在操作中往回走, 往前走。这在跟踪参考文献时特别有用。

文件

测量文件可以被保存和加载。它们以.heap为扩展名被保存。

浏览

你可以在不同的显示屏幕之间切换。有许多不同的类型, 如果你有兴趣, 请看文件。

捕获

测量。然而, 测量目标不能在**Heap Explorer**中改变。目标必须在Unity Profiler或Unity提供的其他工具中改变。保存”将结果保存到文件中并显示出来, 而”分析”则不保存而显示出来。需要注意的是, 与“内存管理器”一样, 测量过程中分配的内存不会被释放。

第3章 剖析工具

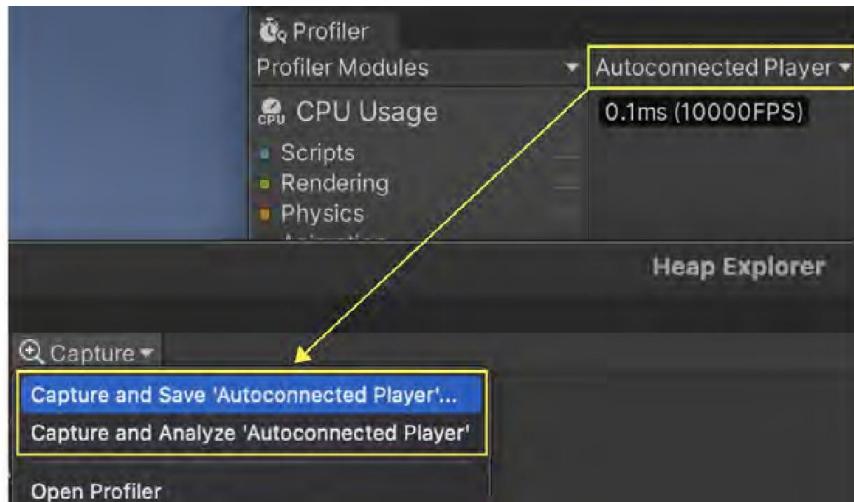


图3.57 切换测量目标

测量结果屏幕看起来像这样这个屏幕被称为“概览”。



图3.58 堆资源管理器测量结果（概览）

在概览中，特别感兴趣的类别是本地内存使用量和托管内存使用量，当按下调查按钮时，它们分别用绿线标记。

3.5 堆资源管理器。

欲了解更多信息, 请[点击这里](#)。

在下面的章节中, 我们将重点讨论显示类别细节的重要方面。

1. 宗旨

如果Native Memory是Investigate, C++对象将出现在这个区域。

对于管理型内存, C#对象会显示在这个区域。

C++ Objects							
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	1	False	True	0x151D107E0	4114
Texture2D	AreaTex	350.5 KB	1	False	True	0x14FF75F50	4066
Texture2D	Large01	256.5 KB	1	False	True	0x15199E440	4092
Texture2D	Medium01	256.5 KB	1	False	True	0x14FF73010	4010

图3.59 对象显示区

头部有几个不熟悉的项目, 应该添加。

ADDOL

Don't Destroy On Load的缩写。它显示该对象是否被指定为一个在场景转换后不被销毁的对象。

持久性

它是否是一个持久性的对象。这是Unity在启动时自动生成的对象。

通过选择图3.59中的对象, 更新以下章节介绍的显示区域。

2. 参考文献。

目标对象被引用的对象被显示出来。

第3章 剖析工具

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

▲图3.60 参考文献。

3. 参考资料：

目标对象所指的对象被显示出来。

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

▲图3.61 参考文献。

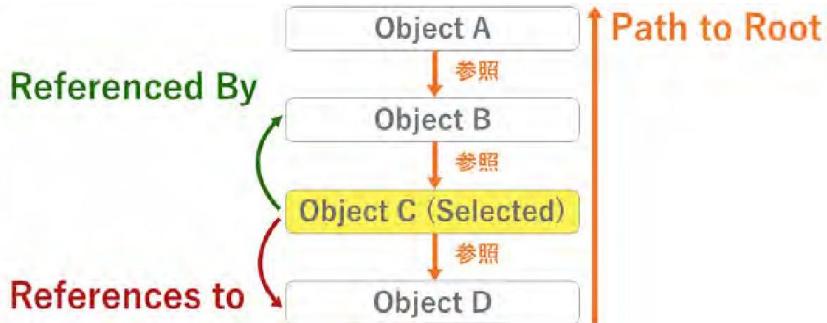
4. 通往根部的路径

引用目标对象的根对象被显示出来。这在调查内存泄漏时很有用，因为你可以看到是什么在持有引用。

2 Path(s) to Root		
Type	C++ Name	Depth Address
UnityEngine.Rendering.RenderPipelineManager	static s_CurrentPipelineAsset	6 0x106F1E
...Universal.UniversalRenderPipelineAsset	UniversalRenderPipelineAsset	0x1519CF
UniversalRenderPipelineAsset	UniversalRenderPipelineAsset	0x1519D2
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

▲图3.62 通往根部的路径。

下面的图片总结了到目前为止的项目。



▲ 图3.63的图像参考。

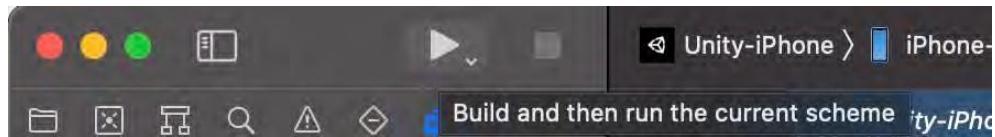
正如到目前为止所介绍的, Heap Explorer提供了一套完整的内存泄漏和内存调查所需的功能。它也非常轻便, 所以请考虑使用这个工具。如果你喜欢它, 最好再加一颗星以示感谢。

3.6 Xcode

Xcode是苹果公司提供的集成开发环境工具;当Unity中的目标平台被设置为iOS时, 构建产品就是Xcode项目。建议使用Xcode进行严格的验证, 因为它提供的数字比Unity更准确。在这一节中, 我们将涉及三种剖析工具:Debug Navigator、GPU Frame Capture和Memory Graph。

3.6.1 简介方法

有两种方法可以从Xcode进行剖析:第一种是直接从Xcode在终端上构建和运行应用程序。如图3.64所示, 只需按下运行按钮即可开始分析。本文件中省略了建筑证书等设置。



▲图3.64 Xcode运行按钮

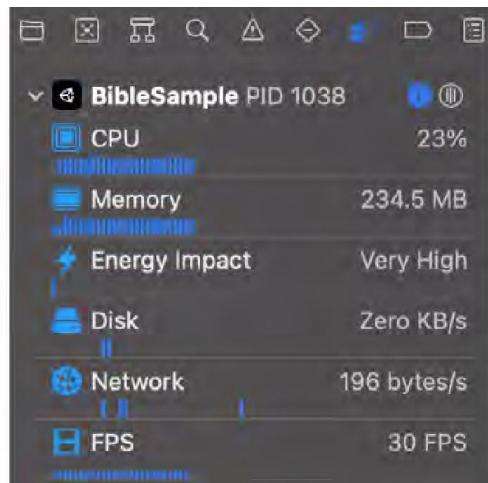
第二种方法是将一个正在运行的应用程序附加到Xcode调试器上。这可以通过在运行应用程序后从Xcode菜单“Debug -> Attach to Process”中选择运行中的进程来进行分析。请注意，构建证书必须是针对开发者的（苹果开发）；你不能用Ad Hoc或企业证书附加。



▲图3.65 Xcode中的调试器附件。

3.6.2 调试导航器

调试导航器允许你仅仅通过从Xcode运行应用程序来检查调试仪表，如CPU和内存。运行应用程序后，按图3.66中的喷雾符号，显示六个项目。或者，从Xcode菜单中，按查看。它也可以通过‘->导航仪->调试’打开。以下各节将对这些项目进行解释。



▲ 图3.66 选择调试导航器

1. CPU测量仪

你可以看到有多少CPU被使用。还可以查看每个线程的使用率。

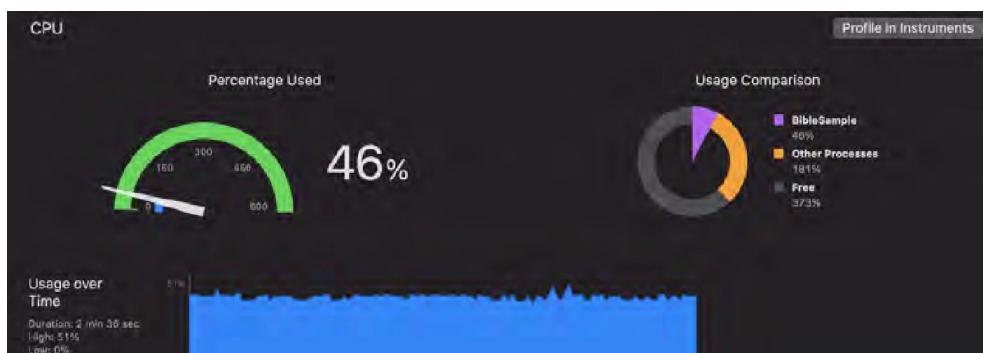


图3.67 CPU测量仪。

2. 记忆仪。

可以查看内存消耗的概况。详细的分析，包括细分，是不可能的。

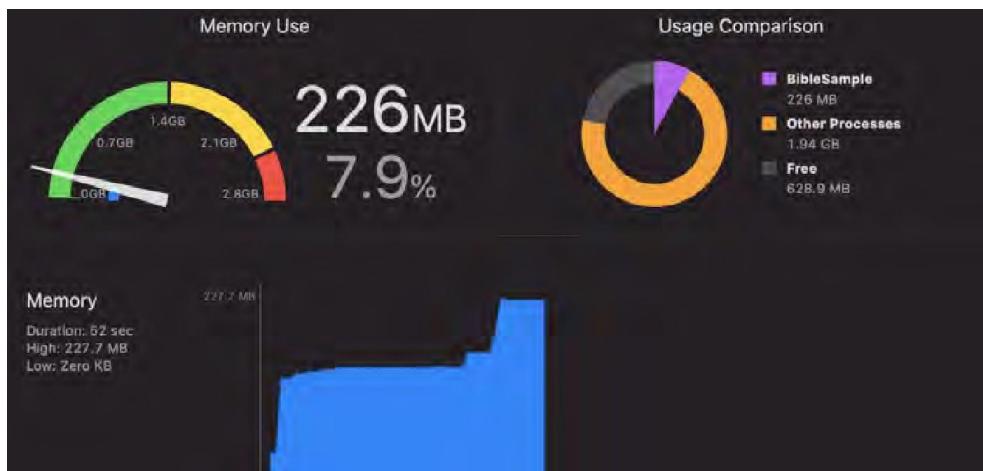


图3.68 内存表。

3. 能量计

你可以看到你的耗电概况，并按CPU、GPU、网络等细分使用。

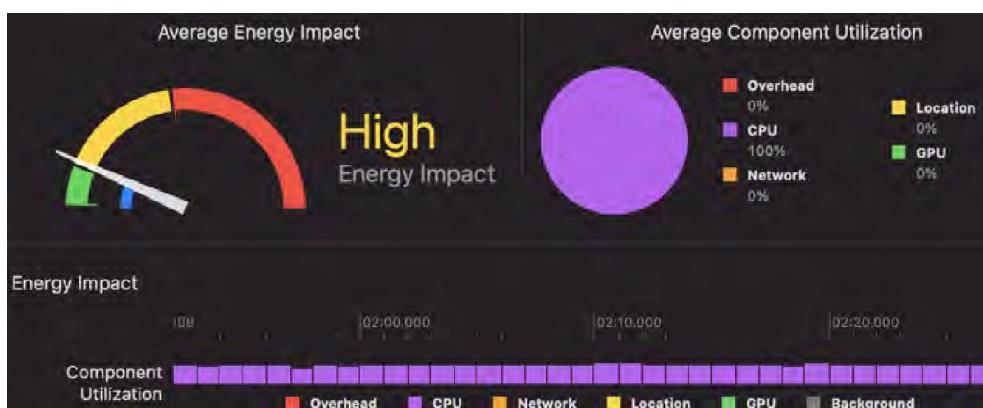


图3.69 能量表。

4. 磁盘测量仪。

文件I/O概述。这对于检查文件是否在意外时间被读取或写入很有用。

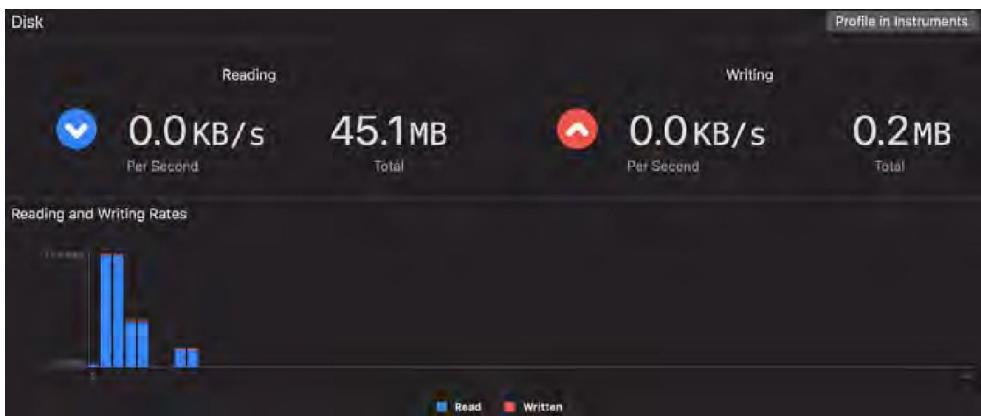


图3.70 磁盘仪。

5. 网络测量仪

你可以看到网络通信的概况，这和磁盘一样，对检查意外的通信很有用。



图3.71 网络仪表。

6. FPS测量仪。

这个仪表在默认情况下是不显示的。当启用3.6.3节中描述的GPU帧捕获时，它就会显示出来，让你不仅可以检查FPS，还可以检查着色器阶段的使用情况以及每个CPU和GPU的处理时间。

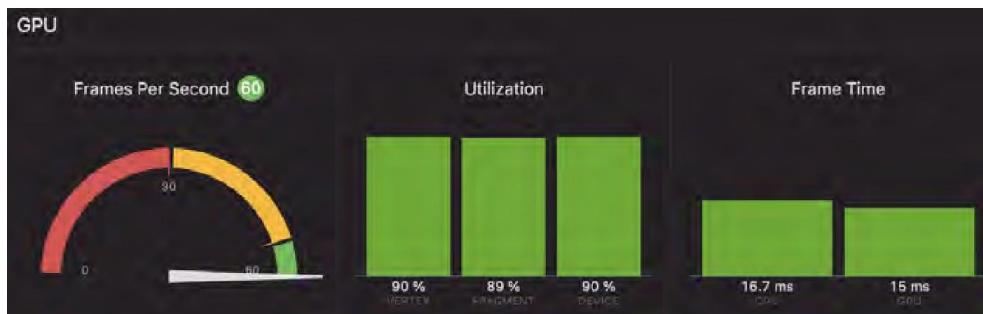


图3.72 FPS表。

3.6.3 GPU帧捕获

GPU Frame Capture是一个允许在Xcode上进行帧调试的工具。像Unity的帧调试器一样，它允许你看到渲染完成之前的过程，由于每个着色器阶段的信息比Unity中的更多，它可能对分析和改进瓶颈很有用。下一节将介绍如何使用它。

1. 准备

要在Xcode中启用GPU Frame Capture，你需要编辑方案。首先，通过选择‘产品->方案->编辑方案’打开方案编辑屏幕。

3.6 Xcode



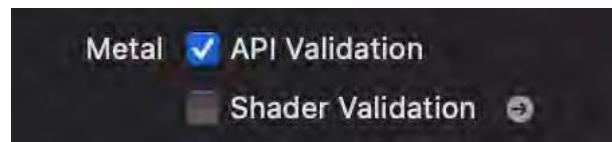
图3.73 计划编辑屏幕。

接下来，从“选项”标签，将GPU帧捕获改为“金属”。



▲ 图3.74 启用GPU帧捕获功能

最后，从“诊断”选项卡，为金属启用“API验证”。



▲ 图3.75 启用API验证

2. 完全自动化的公共图灵测试，将计算机和人类区分开来

在执行过程中，通过按下调试栏中的相机符号来进行捕捉。根据场景的复杂性，第一次捕捉可能需要一些时间，所以要有耐心。请注意，在Xcode13或更高版本中，该图标已被改为金属图标。



图3.76 GPU帧捕获按钮

当捕获完成后，将显示以下摘要屏幕。

3.6 Xcode

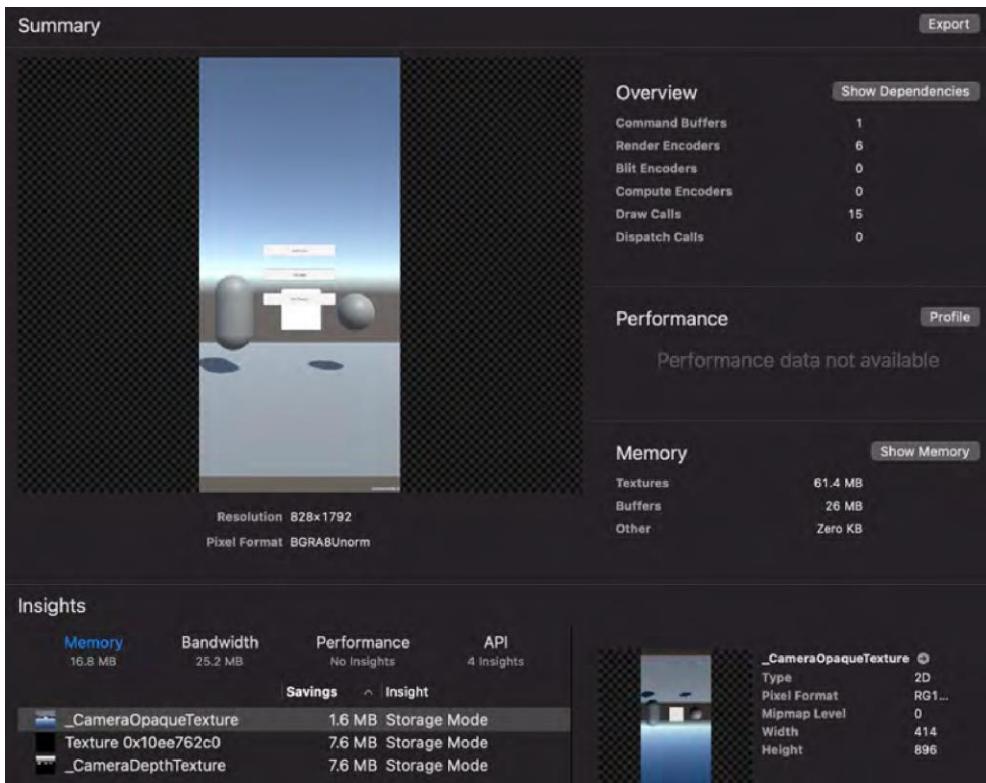
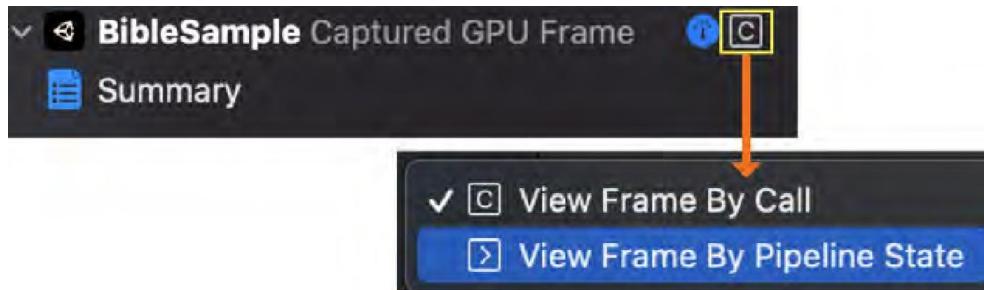


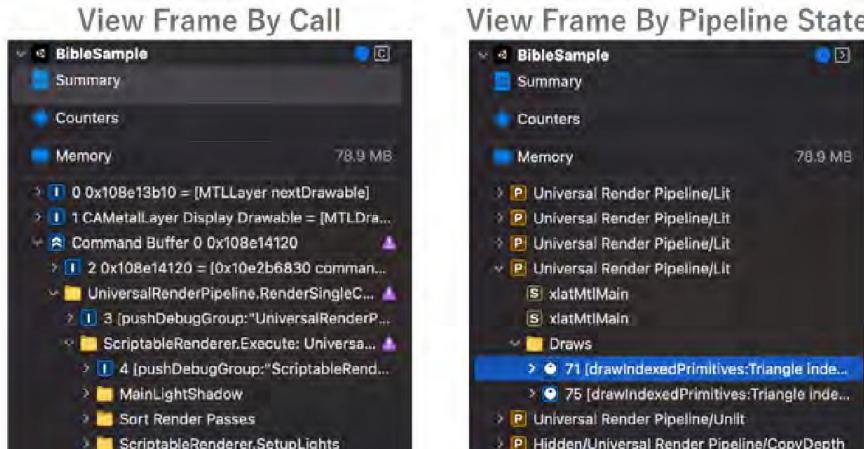
图3.77 摘要屏幕。

从这个摘要屏幕，你可以移动到一个屏幕，在那里你可以检查绘图的依赖性、内存和其他细节。导航区也显示与绘图有关的命令。有两种显示方法：“按调用查看帧”和“按管道状态查看帧”。



▲图3.78 改变显示方式

在 "按调用" 显示中, 所有的绘图命令都按调用的顺序列出。这包括缓冲区的设置和其他绘制的准备工作, 因此有非常多的命令被排在一起。按管道状态只列出了每个着色器所绘制的几何图形的绘制命令。建议根据你所调查的内容, 在不同的显示器之间进行切换。



▲图3.79 显示的差异

可以按下导航区的绘图命令, 查看该命令使用的属性信息。属性信息包括纹理、缓冲器、采样器、着色器功能和几何。每个属性都可以被双击以获得更多信息。例如, 你可以看到着色器代码本身, 以及采样器是Repeat还是Clamp。

3.6 Xcode

Vertex					
B Plane	Index	1 KB	Offset: 0x0	VGlobals	Read
B ScratchBuffer0_1	Buffer 0	4 MB	Offset: 0x240	UnityPerDraw	Read
B ScratchBuffer0_1	Buffer 1	4 MB	Offset: 0x403c0	UnityPerMaterial	Read
B Compute_0	Buffer 2	80 bytes	Offset: 0x0	MainLightShadows	Read
B ScratchBuffer0_1	Buffer 3	4 MB	Offset: 0x4c0	vertexBuffer.0	Read
B Plane	Buffer 4	6 KB	Offset: 0x0		
Geometry	Post Vertex Tran...				
Vertex Attributes	Vertex Attributes				
P xlatMtlMain (Universal Re...	Vertex Function	Library 0x11aaa6a20 [Jus...]			
Fragment					
— Texture 0x114305d50	Texture 0	(Cube) 128 × 128	RGBA8Unorm	unity_SpecCube0	Read
■ UnityWhite	Texture 1	4 × 4	RGBA8Unorm	_BaseMap	Read
• TempBuffer 1 2048x2048	Texture 2	2048 × 2048	Depth32Float	_MainLightShado...	Read

图3.80 绘图命令的细节。

几何属性不仅以表格形式显示顶点信息，而且还允许用户移动摄像机来检查几何体

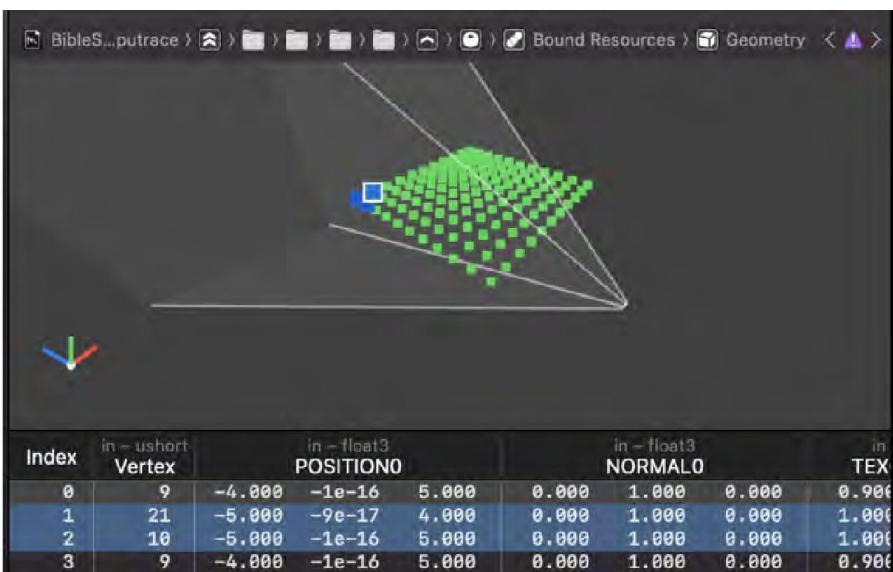


图3.81 几何图形查看器。

第3章 剖析工具

下一节将介绍“摘要”屏幕中“性能”栏的“参数”。点击这个按钮，开始进行更详细的分析。当分析完成后，绘制的时间将显示在导航仪区域。

> 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

▲ 图3.82 剖面图后显示。

分析的结果也可以在“计数器”屏幕上更详细地查看。在这个屏幕上，你可以以图形方式看到每个绘图的处理时间，如顶点、光栅化和片段。



图3.83 计数器屏幕。

下文介绍了“摘要”屏幕中“显示内存”一栏的情况。点击这个按钮将带你到一个屏幕，你可以检查GPU正在使用的资源。显示的信息主要是纹理和缓冲区。检查是否有任何不必要的项目是个好主意。

3.6 Xcode

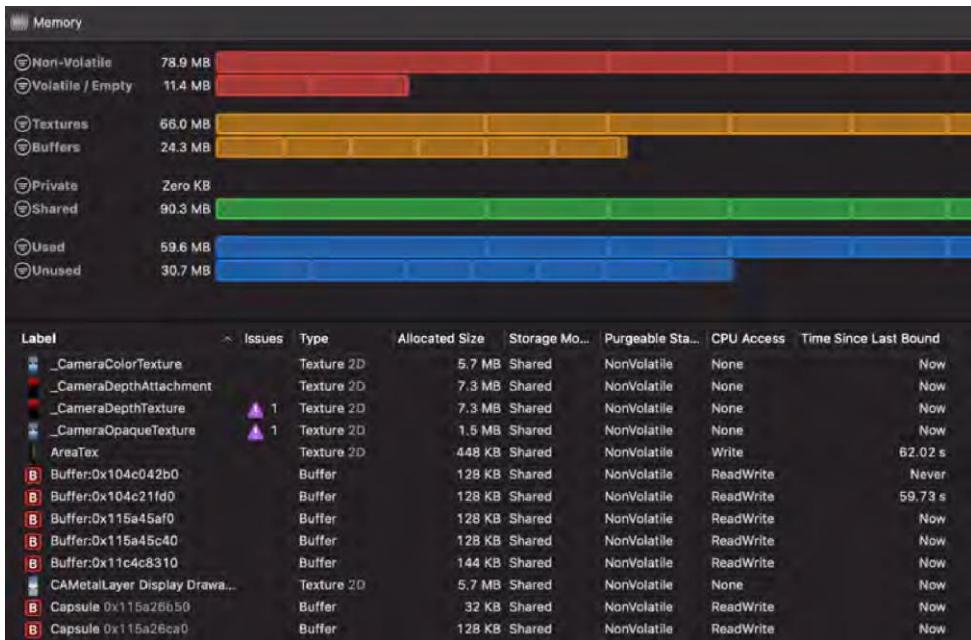
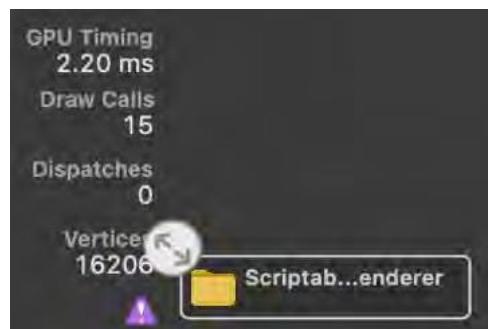


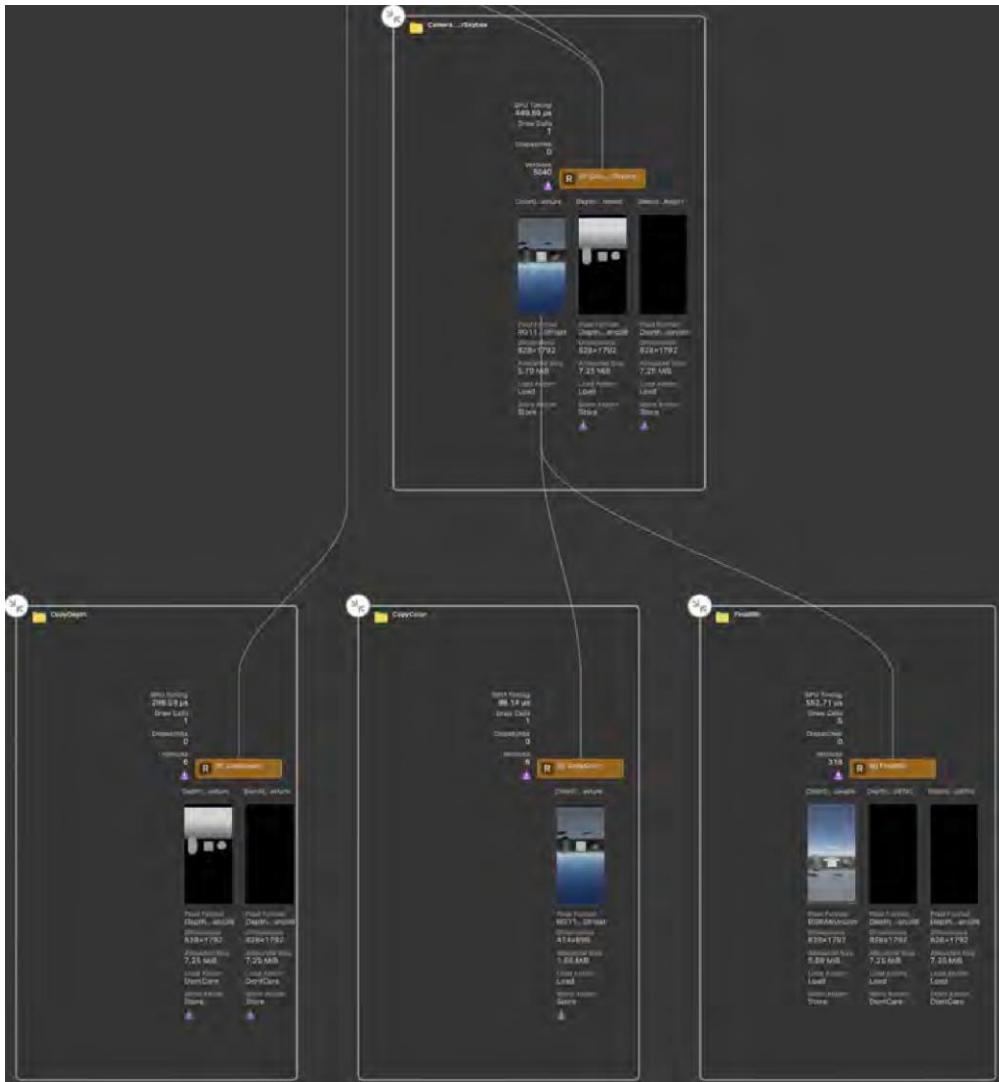
图3.84 GPU资源确认界面。

最后，对“摘要”屏幕的“概览”部分的“显示依赖关系”进行了解释。点击这个按钮将显示每个渲染路径的依赖关系，通过在查看依赖关系时点击“箭头向外”的按钮，你可以打开该级别以下的进一步依赖关系。



▲ 图3.85 打开依赖关系的按钮

当你想看哪些图纸取决于什么时,请使用此屏幕。



▲ 图3.86 打开层次结构

3.6.4 记忆图表

这个工具可以让你分析捕获时机的内存情况。左边的导航区显示实例，通过选择它们，参考关系以图形方式显示。右边的检查器区域显示有关实例的详细信息。

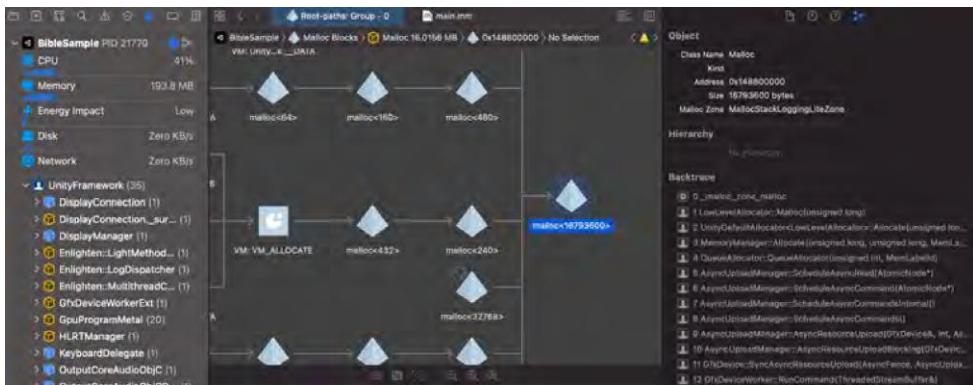


图3.87 MemoryGraph捕获屏幕。

这个工具可以用来调查那些不能在Unity上测量的对象的内存使用情况，例如插件。下面解释如何使用它。

1. 提前准备

需要对计划进行编辑以获得Memory Graph中的有用信息。

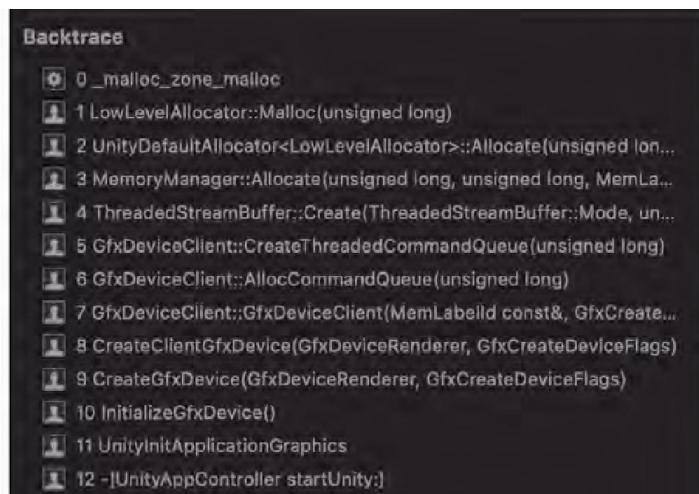
进入“产品->方案->编辑方案”，打开方案编辑界面。然后点击从“诊断”标签，启用“Malloc堆栈记录”。

第3章 剖析工具



▲ 图3.88 启用Malloc栈记录

通过启用这个功能, Backtrace将显示在Inspector中, 你可以看到它是如何分配的。



▲ 图3.89 背面追踪显示。

2. 完全自动化的公共图灵测试, 将计算机和人类区分开来

捕获是在应用程序运行时按下调试栏的一个类似分支的图标来进行的。



图3.90 内存图表捕捉按钮

内存图也可以通过“文件->导出内存图”保存为文件。这个文件可以用vmmap命令、heap命令和malloc_history命令进行更深入的调查。如果你有兴趣，请调查。作为一个例子，vmmap命令的摘要显示如下，使你能够掌握整个情况，这在MemoryGraph中是难以掌握的。

▼ 清单3. 5vmmap摘要命令

1: vmmap --summary hoge.memgraph

REGION TYPE	VIRTUAL SIZE	RESIDENT SIZE	DIRTY SIZE	SWAPPED SIZE	VOLATILE SIZE	NONVOL SIZE	EMPTY SIZE	REGION COUNT
<hr/>								
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	365.9M	143.5M	41.6M	0K	95.6M	13.2M	4362

图3.91 MemoryGraph摘要显示。

3.7 器械

Xcode有一个叫做Instruments的工具，专门用于详细的测量和分析；可以通过选择“产品->分析”来建立仪器。一旦完成，就会打开一个屏幕，选择一个测量模板，如下图所示。

第3章 剖析工具

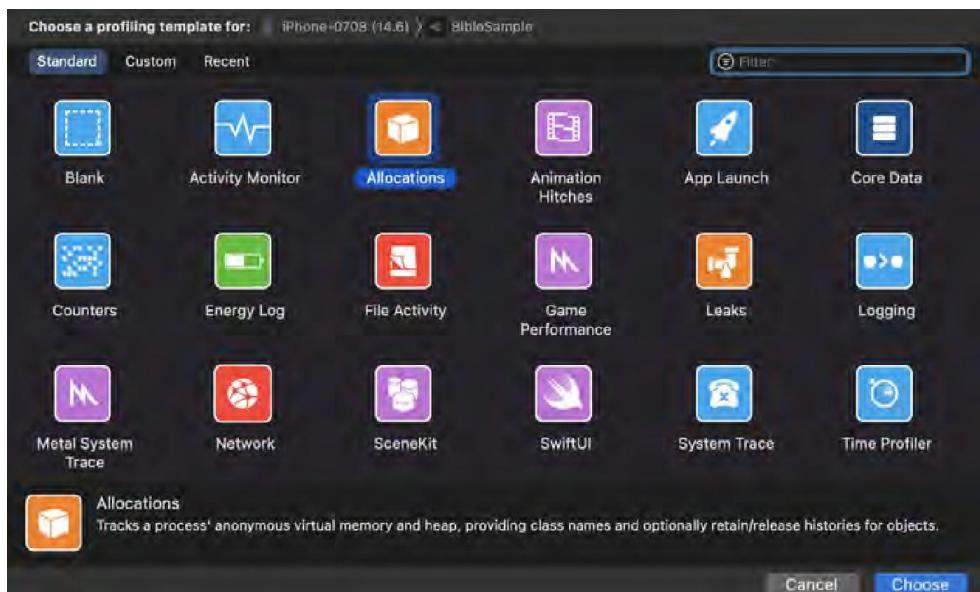


图3.92 仪器模板选择屏幕

从大量的模板中可以看出，仪器可以分析广泛的内容。本节重点介绍其中最常用的“时间规划器”和“分配”。

3.7.1 时代周刊

Time Profiler是一个测量代码执行时间的工具，它在Unity Profiler中。与CPU模块一样，它被用来提高处理时间。要开始测量，必须按下Time Profiler工具条上红色圈出的记录按钮。

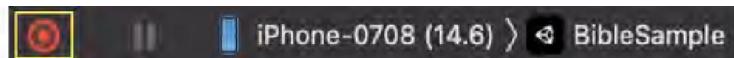


图3.93 记录开始按钮。

当进行测量时，显示屏将看起来像图3.94。

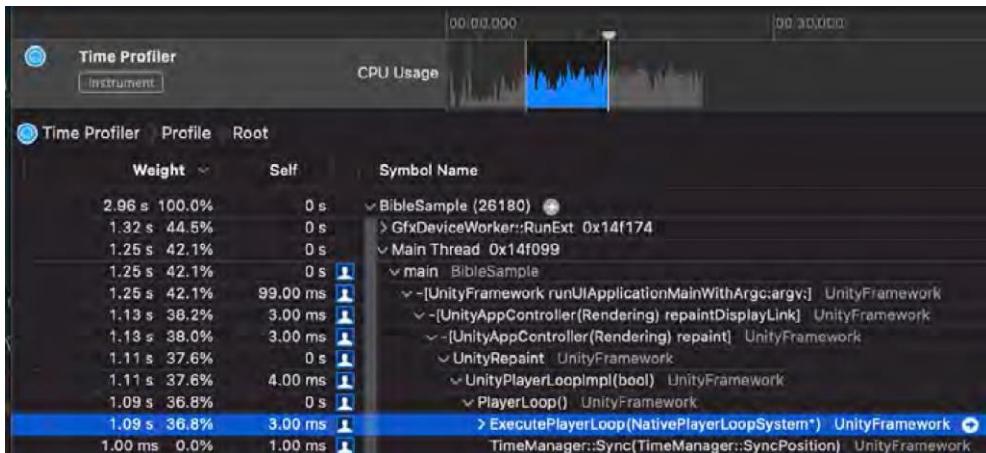
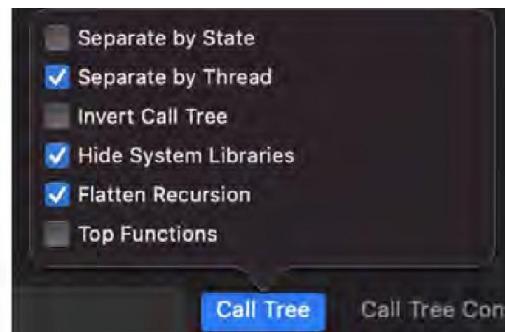


图3.94 测量结果

与Unity Profiler不同的是，分析是分节进行的，而不是一帧一帧地进行。底部的树状视图显示了区间内的处理时间。在优化游戏逻辑的处理时间时，建议在树状视图中分析PlayerLoop下面的处理。为了使树状视图的显示更容易，建议在Xcode的底部设置Call Trees设置，如图3.95所示。特别是，检查隐藏系统库。这使得通过隐藏无法访问的系统代码，更容易进行调查。



▲ 图3.95 呼叫树设置。

第3章 剖析工具

通过这种方式，可以对处理时间进行分析和优化。

时间管理器中的符号名称与统一管理器中的不同。没有大的区别，但符号是“`class_name_function_name_random_string`”。

```

    < RuntimeInvoker_TrueVoid_t700C6383A2A510C2CF4DD86DABD6CA9FF70ADAC5(void (*)(), MethodInfo const*, 
    < SampleScript_Update_mB3FBFF57BD6F501BF7908ECFF223E17CCB509C38 UnityFramework
    > DebugLogHandler_CUSTOM_Internal_Log(LogType, LogOption, ScriptingBackendNativeStringPtrOpaque*, So
    > SampleScript_TestMethod_m390D7F0A04D12D51B91D3A4D94835D6D77DA194F UnityFramework
    > DebugLogHandler_LogFormat_mB876FBE8959FC3D9E9950527A82936F779F7A00C UnityFramework
    > EventSystem_Update_mF0C1580BB2C9A125C27282F471DB4DE6B772DE6D UnityFramework

```

▲ 图3.96 时间管理器中的符号名称

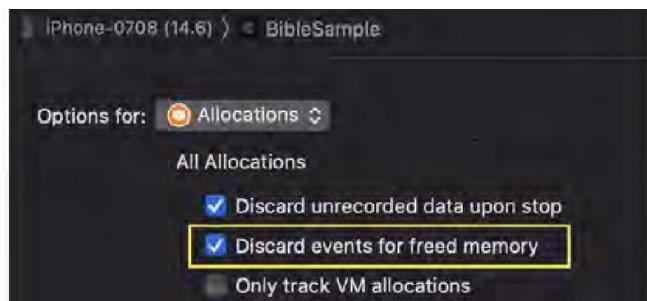
3.7.2 拨款。

Allocations是一个测量内存使用的工具。它被用于内存泄漏和使用情况的改善。



图3.97 分配的测量屏幕。

在你开始测量之前, 打开 “文件 -> 录音选项”, 并勾选 “丢弃自由内存的事件”。



▲ 图3.98 选项设置屏幕。

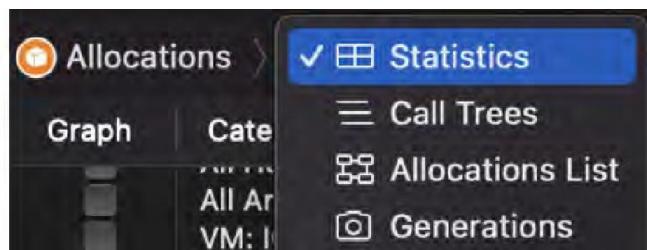
如果这个选项被启用，当内存被释放时，该记录就会被销毁。



▲ 图3.99 根据选项设置而产生的差异

从图3.99中可以看出，有选项和无选项时，外观都有很大变化。有了选项，只有在分配内存时才会记录行数。另外，当分配的区域被释放时，记录的线路会被销毁。换句话说，设置这个选项意味着如果一个行仍然在内存中，它没有被从内存中释放出来。例如，如果设计是通过场景转换来释放内存，那么如果在转换之前有许多行留在场景部分，就可能怀疑有内存泄漏。在这种情况下，请关注树状视图中的细节。

屏幕底部的“树状视图”显示指定范围的细节，与“时间编辑器”类似。有四种不同的方式来显示这个树状视图。



▲ 图3.100 选择一种显示方法

最推荐的显示方法是呼叫树。这使你能够跟踪哪个代码引起的分配。呼叫树显示选项在屏幕的底部，你可以设置隐藏系统库等选项，其方式与图3.95中介绍的时间管理器相同。图3.101捕捉到了调用树的显示，显示出12.05MB的分配是由SampleScript的OnClicked产生的。

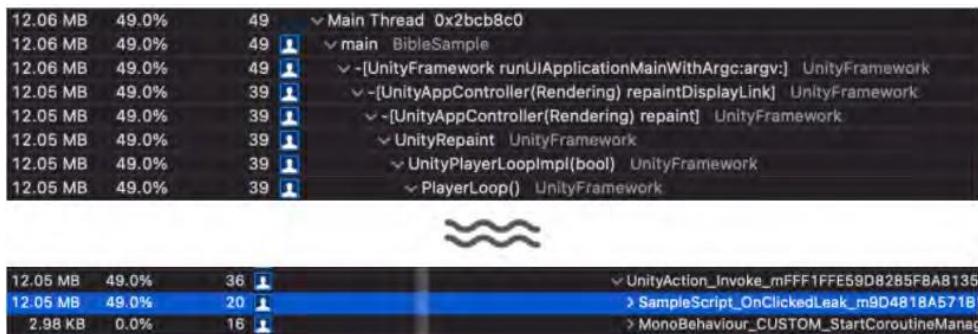


图3.101 呼叫树显示。

最后，我们介绍一个叫做“生成”的功能：在Xcode的底部有一个叫做“标记生成”的按钮。



▲ 图3.102 标记生成按钮

第3章 剖析工具

按这个键将存储当时的记忆。然后你可以再次按下标记

按“生成”键记录新分配的内存量与之前的数据相比。

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

▲图3.103 世代。

图3.103中的每一代在详细查看时都以呼叫树的形式显示，因此可以跟踪导致内存增加的原因。

3.8 安卓工作室

Android Studio是一个集成的Android开发环境工具。这个工具可以用来衡量应用程序的状态。四个可配置的项目是CPU、内存、网络和能源。本节首先介绍了剖析方法，然后介绍了CPU和内存的测量项目。

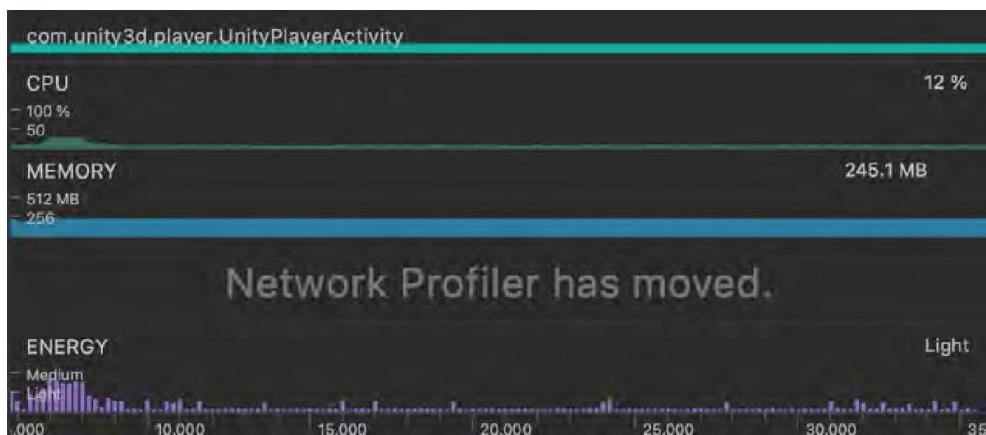


图3.104 简介屏幕。

3.8.1 简介方法

有两种方法可以进行剖析：第一，通过Android Studio构建和剖析。在这种方法中，你首先从Unity导出Android Studio项目，在构建设置中勾选“导出项目”，然后构建。



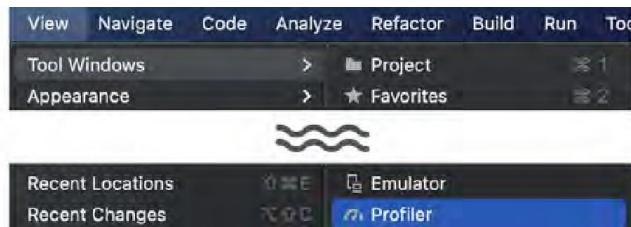
图3.105 导出一个项目

接下来，在Android Studio中打开导出的项目。然后，在连接安卓设备的情况下，按右上角类似仪表的图标，开始构建。构建完成后，应用程序将启动，配置文件将开始。



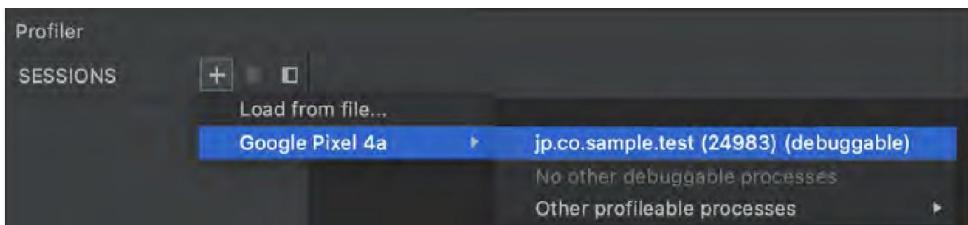
图3.106 序列启动图标。

第二种方法是将运行中的进程附加到调试器上，并对其进行测量。首先，从Android Studio菜单“视图->工具窗口->Profiler”中打开Android Profiler。



▲ 图3.107 打开安卓系统编辑器

接下来，从打开的Profiler中的**SESSIONS**中选择要测量的会议。要连接一个会话，要测量的应用程序必须正在运行。此外，二进制文件必须从开发构建中改编。一旦会话被连接，简介就开始了。



▲ 图3.108 选择要剖析的SESSION

第二种附加到调试器的方法值得记住，因为它不需要导出项目，而且是现成的。

严格来说，你需要在AndroidManifest.xml中设置debug-gable和profileable，而不是在Unity的Development Build中。在Unity中，如果你进行Development Build，那么debuggable被设置为true。

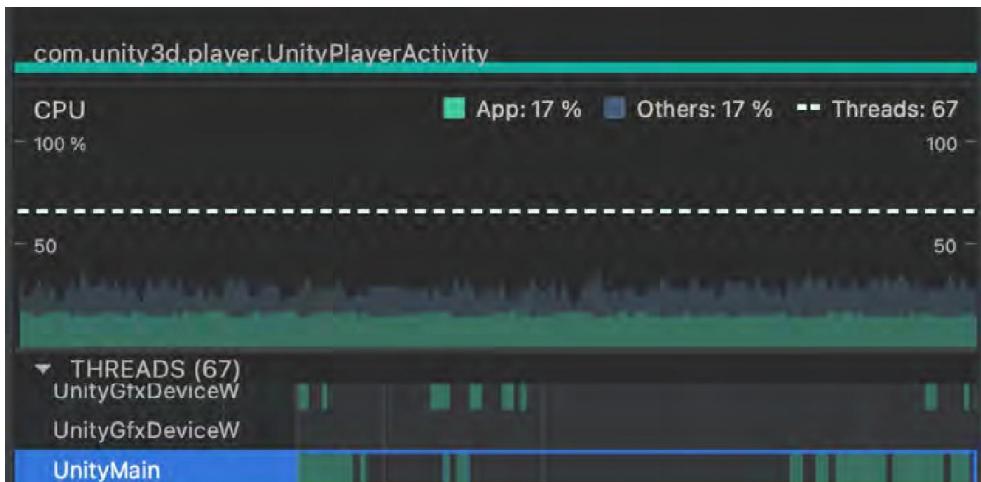
3.8.2 CPU测量

CPU测量屏幕看起来像图3.109。仅仅这个屏幕并不能告诉你什么东西消耗了多少处

第3章 剖析工具

理时间。要获得更详细的视图,请点击[s](#)

必须选择红色。



▲ 图3.109 CPU测量顶部屏幕，线程选择

在选择了一个线程后，通过按下记录按钮来测量该线程的callstack。如图3.110所示，有几种测量类型可供选择，但“Callstack Sample Recording”应该是可以的。

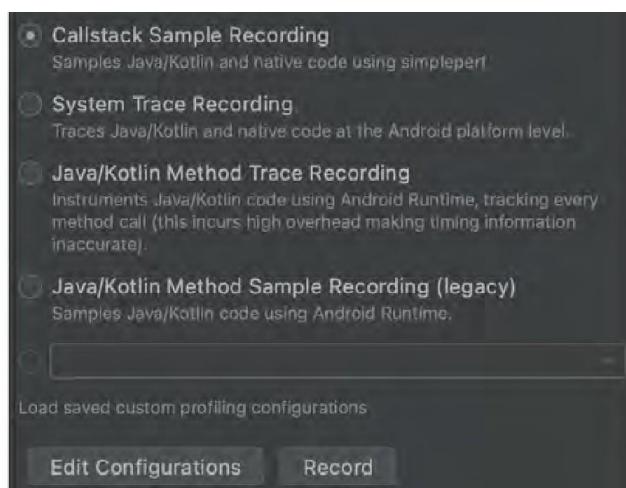


图3.110 记录开始按钮。

第3章 剖析工具

按下“停止”按钮就可以完成测量并显示结果。结果屏幕显示它看起来就像Unity Profiler中的一个CPU模块。

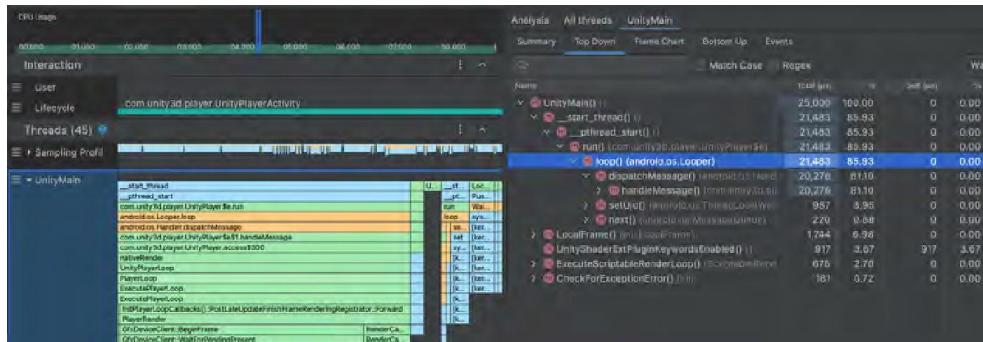


图3.111 呼叫堆栈测量结果屏幕。

3.8.3 记忆测量

内存测量屏幕如图3.112所示。在这个屏幕上不能看到内存分解。

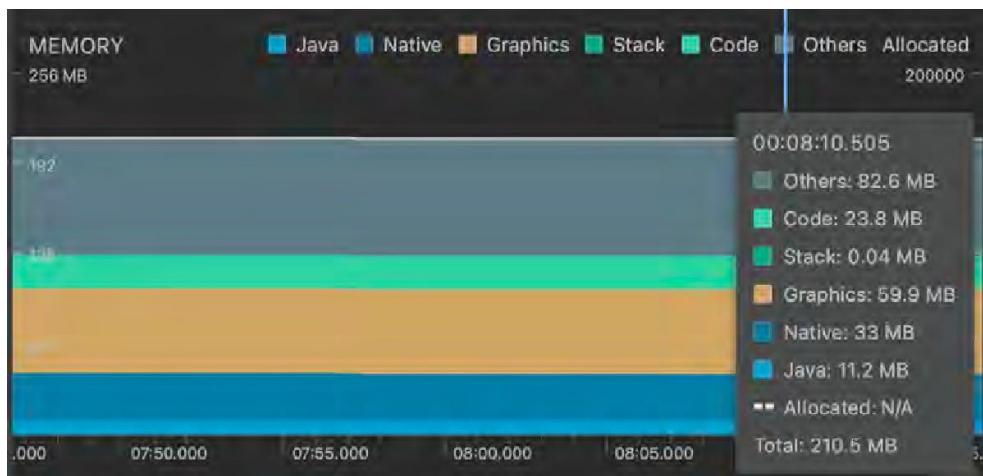


图3.112 内存测量屏幕。

如果你想看到内存的细分，你需要进行额外的测量。有三种测量方法

“捕获堆转储”功能允许在按下按钮的时候获取内存信息。“捕获堆转储”允许在按下按钮的时候获取内存信息。其他按钮是用来分析测量部分的分配。

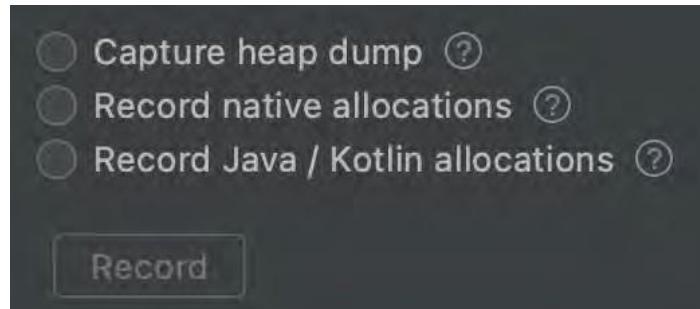


图3.113 内存测量选项。

作为一个例子，图3.114捕获了一个堆倾倒的测量结果。详细的分析会比较困难，因为颗粒度有点粗大。

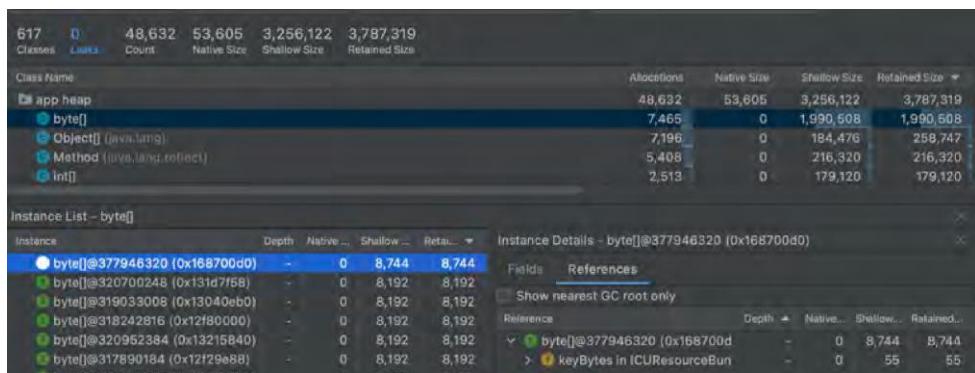


图3.114 堆栈倾倒结果。

3.9 RenderDoc

RenderDoc是一个开源的、高质量的图形调试器工具，可以免费使用。该工具目前支持Windows和Linux，但不支持Mac。支持的图形API是

Vulkan、OpenGL（ES）、D3D11和D3D12。因此，它可以在Android上使用，但不能在iOS上使用。

在这一节中，我们将实际剖析一个Android应用程序。然而，请注意，对安卓系统进行分析有一些限制。首先，安卓操作系统版本必须是6.0或更高。而且要测量的应用程序必须是可调试的。如果在构建时选择了“开发构建”，这就没有问题了。请注意，该简介所使用的RenderDoc的版本是v1.18。

3.9.1 测量方法

首先，准备RenderDoc。从官方网站^{*3}下载安装程序并安装该工具。安装后，打开RenderDoc工具。

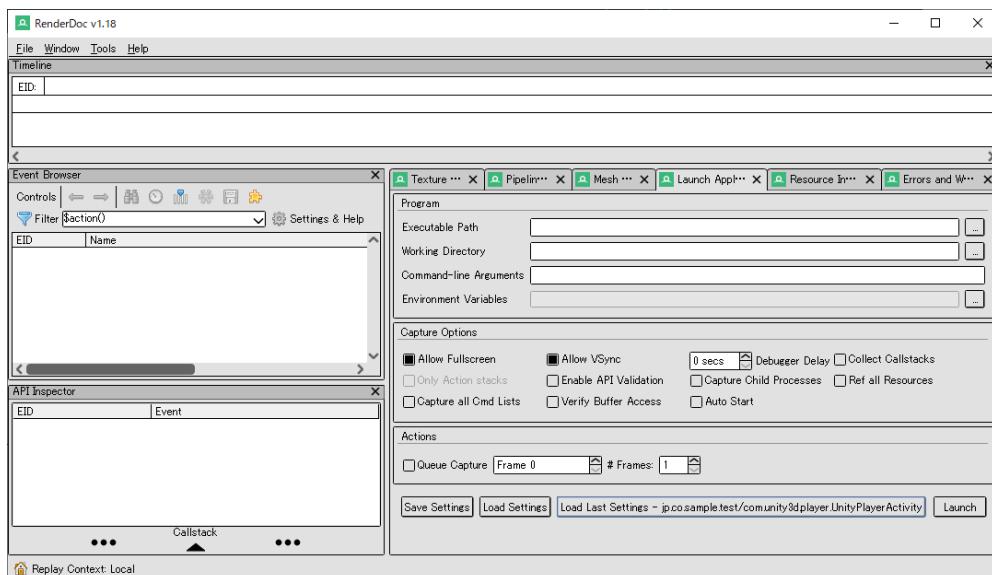


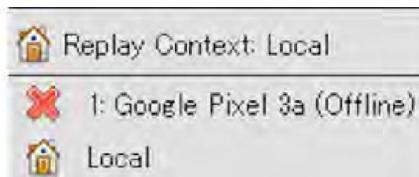
图3.115 启动RenderDoc后的屏幕。

接下来，将你的安卓设备连接到RenderDoc。按左下角的房子符号，显示连接到PC的设备列表。从列表中，选择被测量的

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

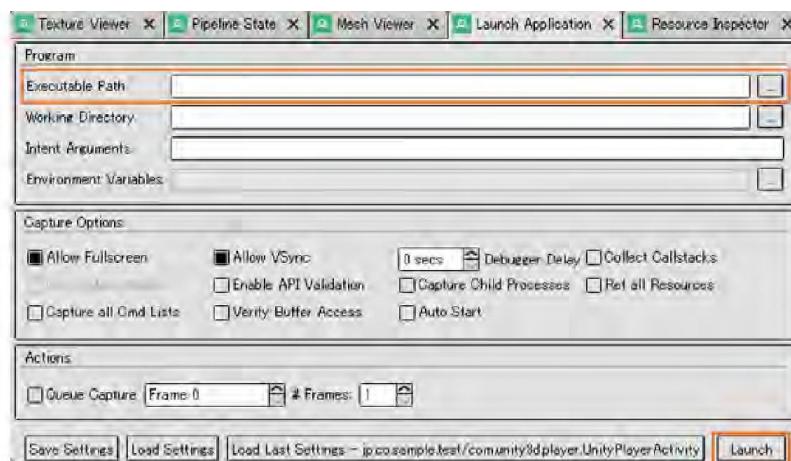
第3章 剖析工具

请选择你要使用的设备。



▲ 图3.116 连接到设备

接下来，选择你想从连接的设备上启动的应用程序。从右侧的标签中选择启动应用程序，并从可执行路径中选择你要运行的应用程序。



▲ 图3.117 启动应用程序标签 选择正在运行的应用程序

当文件浏览器窗口打开时，寻找这次要测量的Pacakge名称并选择活动。

选择以下内容。

3.9 RenderDoc.

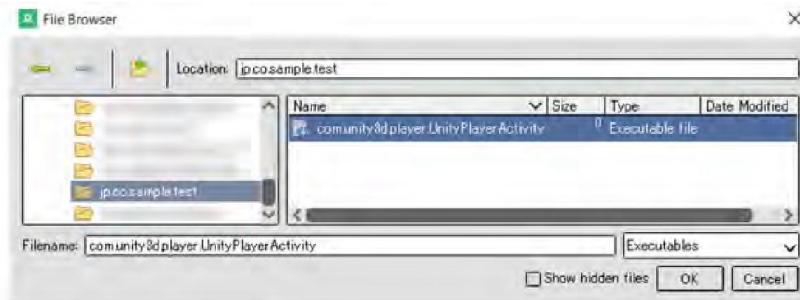
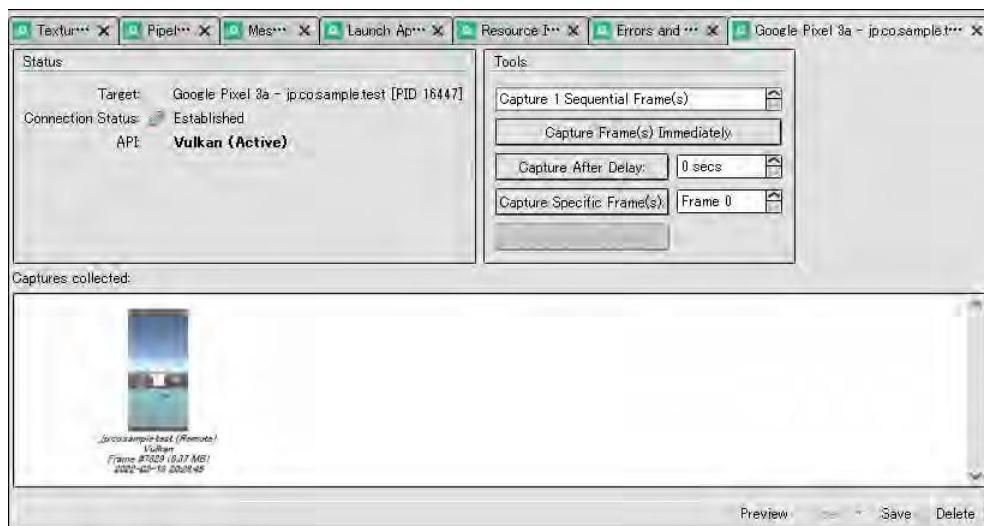


图3.118 选择要测量的应用程序

最后,按下“启动应用程序”中的“启动”按钮将在设备上启动该应用程序。此外,在RenderDoc上,还增加了一个新的标签,用于测量。



▲ 图3.119 用于测量的标签

如果你按下立即捕获帧(s)作为试验,帧数据将被捕获并排成一列。双击该数据以打开捕获的数据。

3.9.2 如何查看捕获数据

RenderDoc有广泛的功能，但在本节中，我们将集中讨论功能中最重要的部分。首先，在屏幕的顶部显示所捕获的帧的时间线。你可以直观地看到每条绘图命令是按照什么顺序执行的。

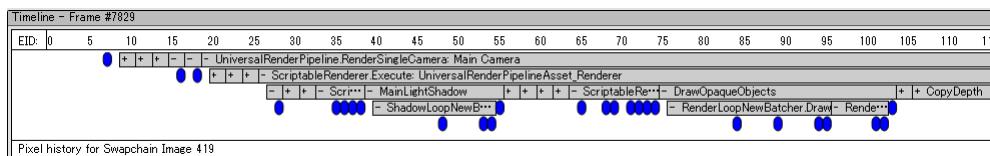


图3.120 时间线。

接下来是事件浏览器。这里按从上到下的顺序列出了每条命令。

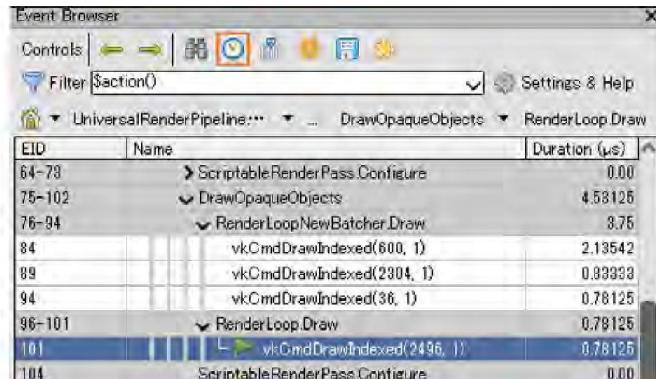


图3.121 事件浏览器。

在事件浏览器中按顶部的“时钟符号”，在“持续时间”中显示每个命令的处理时间。处理时间根据测量时间的不同而不同，所以它只能被视为一个粗略的指导。另外，对DrawOpaqueObjects命令的细分显示，有三个是批量处理的，只有一个是非批量处理的。

下文介绍了每个窗口的情况，这些窗口在右侧的标签中排列。在这些标签中，你会发现

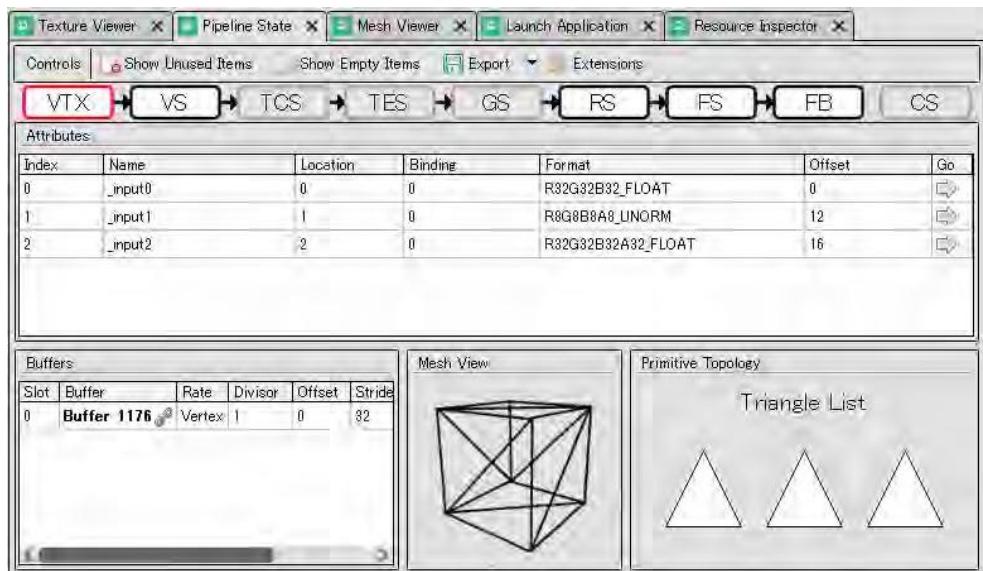
第3章 剖析工具

特别重要的是网格浏览器、纹理浏览器和管道状态。



图3.122 每个窗口。

首先，让我们来谈谈管道状态，在这里你可以看到在物体被绘制到屏幕之前，每个着色器阶段使用了哪些参数。你还可以查看所使用的着色器及其内容。



▲图3.123 管线状态。

在管道状态下显示的阶段性名称是缩写，因此代表官方名称。
总结于3.7。

▼ 表3.7 PipelineState的正式名称

舞台名称	正式名称
VTX	顶点输入
versus (vs, v.)	顶点着色器
TCS	镶嵌控制着色器
TES	镶嵌评估着色器
ÂÂÂ	几何图形着色器
ÂÂÂ	光栅器
雇员	片段着色器
基金会	帧缓冲器
CS	计算着色器

在图3.123中，选择了VTX阶段，在这里你可以看到拓扑结构和顶点输入数据。其他细节，如输出目标纹理状态和混合状态，可以在图3.124的FB阶段看到。

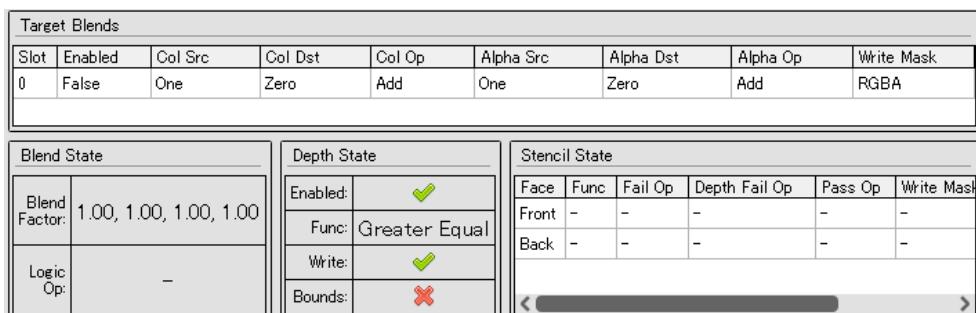


图3.124 FB (帧缓冲器)的状态

图3.125中的FS阶段也允许用户看到碎片着色器中使用的纹理和参数。

第3章 剖析工具

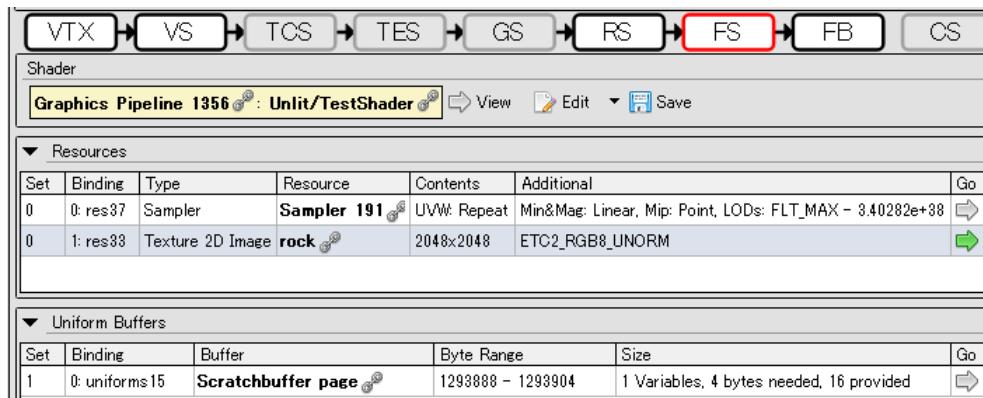


图3.125 FS的状态（碎片着色器）。

在FS阶段中间的资源显示了所使用的纹理和采样器。在FS阶段底部的统一缓冲区部分，显示了CBuffer。这个CBuffer包含数字属性，如平面和颜色。每个项目的右边都有一个“前进”的箭头图标，可以按下它来查看数据的更多细节。

所用的着色器显示在FS阶段的上部；按“查看”可以看到着色器代码。推荐GLSL用于Disassembly类型，以使显示更容易理解。

The screenshot shows the "Disassembly" tab. At the top, there is a dropdown menu labeled "Disassembly type: GLSL (SPIRV-Cross)". The code listed is:

```
1 #version 450
2
3 layout(set = 1, binding = 0, std140) uniform _13_15
4 [
5     float _m0;
6 } _15;
7
8 layout(set = 0, binding = 1) uniform mediump texture2D _33;
9 layout(set = 0, binding = 0) uniform mediump sampler _37;
```

▲ 图3.126 检查着色器代码。

接下来是网格浏览器。这允许你直观地查看网格信息，对优化和调试非常有用。

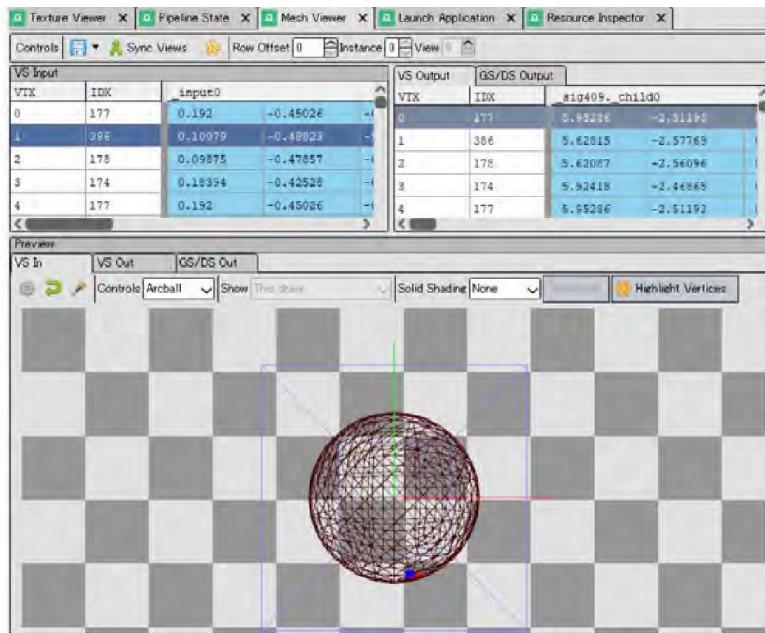


图3.127 网格浏览器。

网格浏览器的上部以表格形式显示网格顶点信息。下部有一个预览屏幕，可以通过移动相机来检查形状。两者都有单独的输入和输出标签，所以你可以看到转换前后的数值和外观的变化。

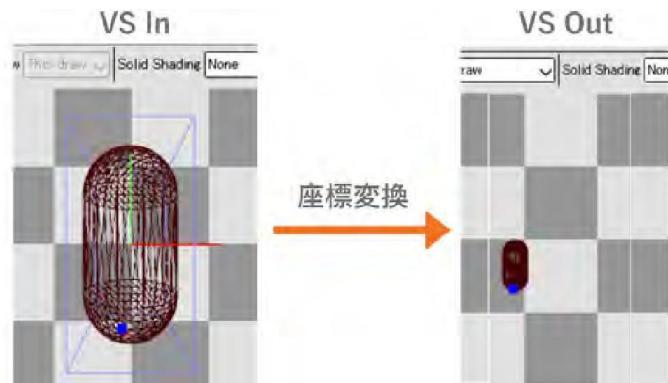


图3.128 网格浏览器中In和Out的预览显示。

最后, 还有纹理查看器。这个屏幕显示在事件浏览器中选择的命令的 ”用于输入的纹理”和 ”输出结果”。

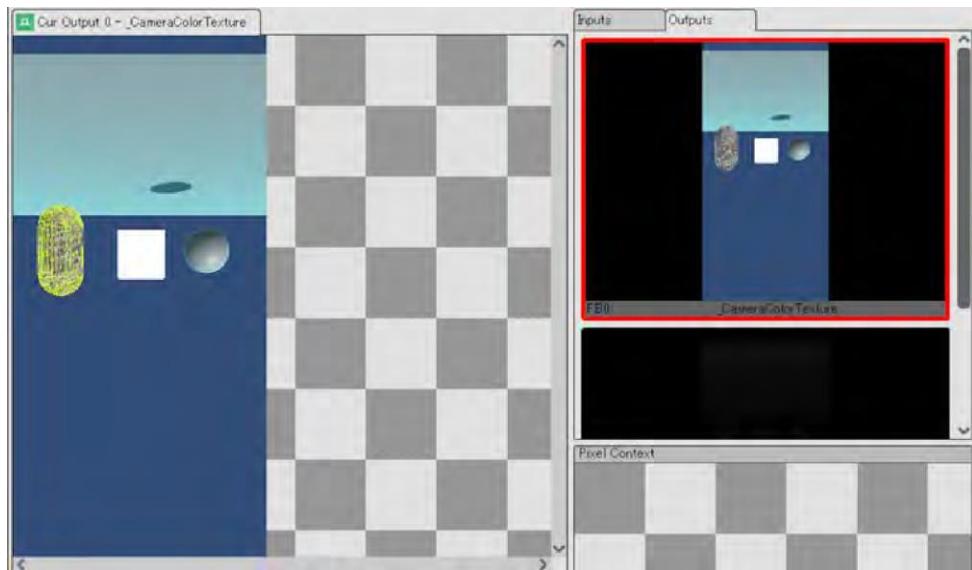


图3.129 纹理查看器的纹理验证屏幕。

在屏幕的右侧区域, 你可以看到输入和输出的纹理。按下所显示的纹理将在屏幕的左侧区域反映出来。屏幕的左侧不仅可以显示, 还可以过滤颜色通道和应用工具栏设置。

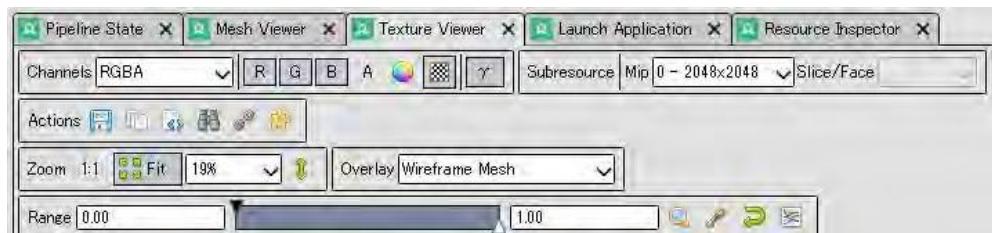


图3.130 纹理查看器工具条。

在图3.129中, 'Wireframe Mesh'被选为Overlay, 所以用这个命令绘制的对象有一个黄色的线框显示, 视觉上更容易理解。

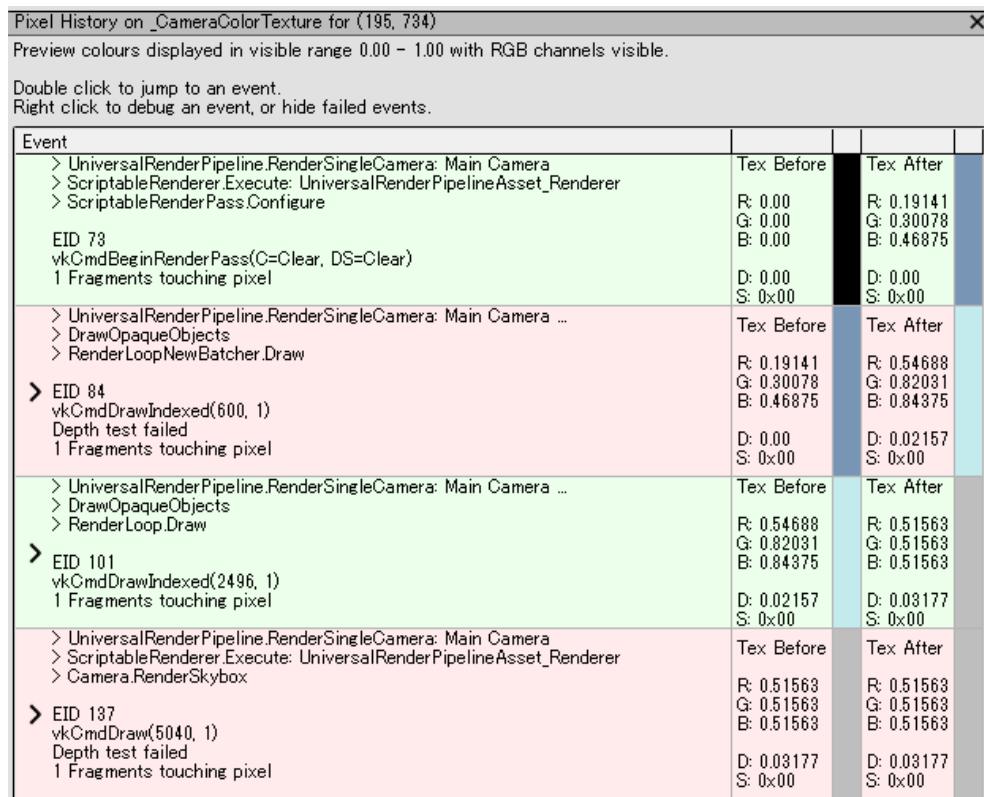
纹理查看器也有一个叫做“像素背景”的功能。该功能允许你查看所选像素的绘制历史。历史记录意味着你可以看到一个像素被填充的频率。这是一个调查和优化超限的有用功能。然而，它只是在每个像素的基础上，所以它不适合在全局基础上调查过度绘制。要调查，在图3.129左侧的屏幕上右击你要调查的区域，位置会反映在像素上下文中。



图3.131 对像素背景的反思

接下来，按下像素上下文中的历史按钮，可以看到该像素的绘制历史。

第3章 剖析工具



▲ 图3.132 像素绘制历史

在图3.132中，有四段历史。绿线表示所有的管道测试，如深度测试，已经通过，管道已经绘制完毕。如果一些测试失败，没有被画出来，则为红色。在捕获的图像中，屏幕清除过程和胶囊绘制都是成功的，而平面和天盒则未能通过深度测试。



性能调整

CHAPTER

0

第四章。

调试实践--资产

游戏制作涉及处理大量不同类型的资产，如纹理、网格、动画和声音。因此，本章总结了有关这些资产的实用知识，包括调整性能时需要注意的设置。

4.1 纹理

图像数据是纹理的来源，是游戏制作的重要组成部分。另一方面，内存消耗相对较高，所以设置必须适当。

4.1.1 进口设置

图4.1显示了Unity中的纹理导入设置。

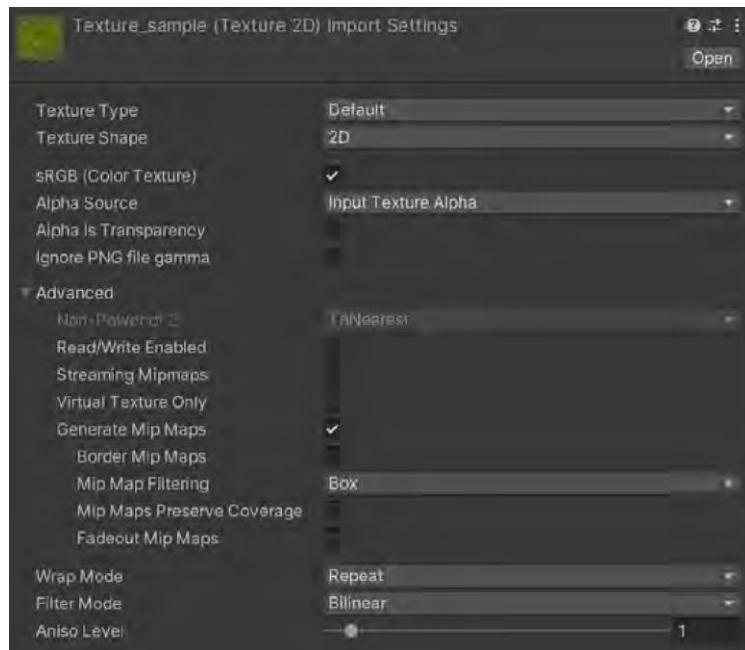


图4.1 纹理设置

以下是需要注意的一些最重要的情况。

4.1.2 读/写

该选项在默认情况下是禁用的。如果禁用，纹理只被部署到GPU内存。如果启用，它们不仅被复制到GPU内存，而且被复制到主内存，因此消耗的数量是两倍。因此，如果你不使用 `Texture.GetPixel` 或 `Texture.SetPixel` 等API，只在Shader中访问它们，请确保禁用它们。

如清单4.1所示，通过设置`makeNoLonger-Readable`为true，也可以避免运行时生成的纹理被复制到主内存中。

▼ 清单 4.1配置 makeNoLongerReadable

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

纹理从GPU内存传输到主内存是很耗时的，所以如果可以读取的话，通过在两者中部署纹理可以提高性能。

4.1.3 生成Mip地图

启用Mip Map设置会使纹理内存的使用量增加约1.3倍。这个设置一般用于三维物体，以减少锯齿和远处物体的纹理传输；对于二维精灵和用户界面图像来说，它基本上是不必要的，所以它应该被禁用。

4.1.4 阿尼索水平

Aniso Level是一个允许以浅角度绘制物体的功能，而不会使纹理的外观变得模糊。这一功能主要用于延伸较远的物体，如地面或地板；Aniso Level值越高，它提供的好处越多，但处理成本也越高。

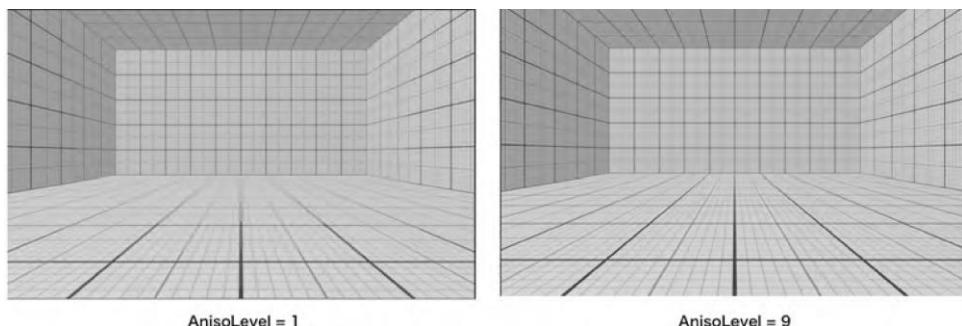


图4.2 Aniso水平适应图像

设定的数值从0到16，但规格有点特殊。

- 0: 无论项目设置如何, 总是禁用
- 1: 基本上禁用。但是, 如果项目设置为强制开启, 则数值被钳制在9~16之间。
- 否则 : 设置为该值

当纹理被导入时, 该值默认被设置为1。因此, 除非你使用高规格的设备, 否则不推荐强制开启; 强制开启可以在“项目设置->质量”下的各向异性纹理中设置。



▲图4.3 强制开启设置。

检查无效的对象是否启用了Aniso级别设置, 或者对于有效的对象是否设置得过高

◦

Aniso Level的效果不是线性的, 而是分阶段切换行为。笔者验证过, 它的变化分为四个阶段:0-1、2-3、4-7和8以后。

4.1.5 压缩设置

纹理应该被压缩, 除非有特定的理由不这样做。如果项目中出现了未压缩的纹理, 可能是人为错误或超出了规定。尽快检查这个问题。关于压缩设置的更多信息, 见2.3.3压缩图像——节。

建议使用TextureImporter来自动进行这些压缩设置, 以避免人为错误。

▼清单4. 2TextureImporter自动化实例

```
1: 使用UnityEditor。  
2:  
3: public class ImporterExample : AssetPostprocessor  
4: {  
5:     private void OnPreprocessTexture()  
6:     {  
7:         var importer = assetImporter as TextureImporter;  
8:         importer.isReadable = false; // 读/写设置等可能 9:  
10:        var settings = new TextureImporterPlatformSettings();  
11:        // Android = "Android", PC = "Standalone"  
12:        settings.name = "iPhone";  
13:        settings.overridden = true;  
14:        settings.textureCompression = TextureImporterCompression.Compressed;  
15:        settings.format = TextureImporterFormat.ASTC_6x6; // 指定压缩格式  
16:        importer.SetPlatformTextureSettings(settings);  
17:    }  
18: }
```

另外，不是所有的纹理都需要采用相同的压缩格式。例如，在用户界面图像中，具有整体梯度的图像很可能会因为压缩而出现明显的质量损失。在这种情况下，建议只对部分目标图像设置较低的压缩率。相反，3D模型等纹理不太可能出现质量损失，因此建议找到一个合适的设置，如高压缩比。

4.2 网络。

下面是在Unity中处理导入的网格（模型）时需要考虑的一些要点。导入的模型数据的性能可以根据设置来改善。应注意以下四点。

- 启用了读/写功能
- 顶点压缩
- 网眼压缩
- 优化网格数据

4.2.1 启用了读/写功能

第一个Mesh音符是读/写启用。模型检查器中的这个选项默认是禁用的。

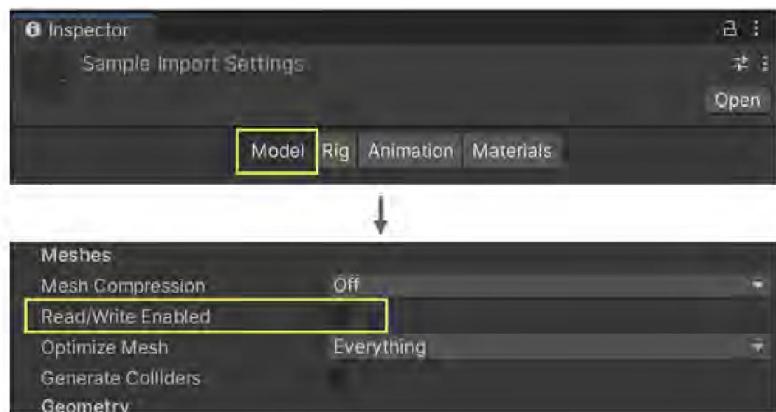


图4.4 读/写设置。

如果你不需要在运行期间访问网格，请将其关闭。具体来说，如果模型被放置在Unity上，只用来播放AnimationClip，可以禁用读写功能而不会有任何问题。

如果启用，CPU可访问的信息被保存在内存中，从而消耗两倍的内存。请通过简单地禁用它来检查内存的节省情况。

4.2.2 顶点压缩

顶点压缩是一个选项，可以将网格顶点信息的精度从平面变为半平面。这减少了运行时的内存使用和文件大小。它可以在“其他”中的“项目设置->播放器”中设置，默认设置如下。



图4.5 顶点压缩的默认设置。

然而,请注意,如果满足以下条件,该顶点压缩功能将被禁用

- 启用读/写功能。
- 启用网格压缩
- 启用动态批处理并可适应的网格(少于300个顶点和少于900个顶点属性)。

4.2.3 网眼压缩

网格压缩允许你改变网格的压缩率。压缩率越高,文件大小就越小,所需的存储量就越少。压缩的数据在运行时被解压缩。这对运行时的内存使用没有影响。

在 Mesh Compression 中的压缩设置有四个选项。

- 关 : 无压缩。
- 低 : 低压缩量
- 中度 : 中度压缩。
- 高 : 高压缩

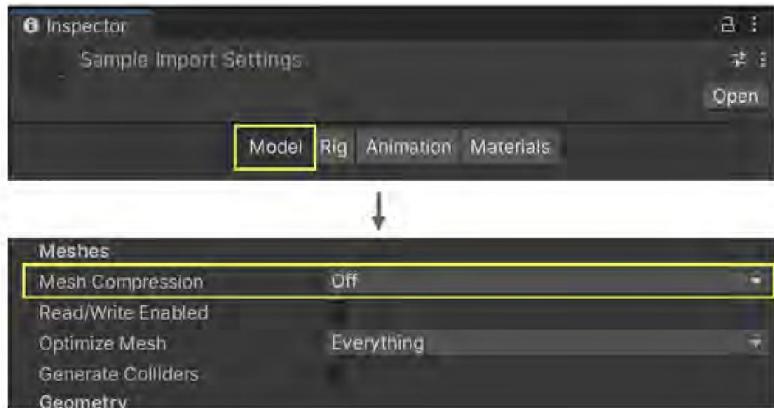


图4.6 网格压缩。

正如第4.2.2节顶点压缩中提到的，启用该选项可以禁用顶点压缩。特别是在有严格内存使用限制的项目中，请在设置该选项前注意这一缺点。

4.2.4 优化网格数据

优化网格数据会自动删除网格中不必要的顶点数据。不必要的顶点数据是由使用中的着色器自动确定的。这具有在运行时减少内存和存储的效果。这可以在“项目设置”->“播放器”的“其他”中设置。



图4.7 优化网格数据设置。

然而，虽然这个选项很方便，因为顶点数据会被自动删除，但要注意，它可能会引起意想不到的问题。例如，在运行时在材料和着色器之间切换时，所访问的属性可能会被删除，从而导致不正确的绘图结果。在其他情况下，当资产包中只使用Mesh时，如果Material设置不正确，可能会被判定为不必要的顶点数据。这在只有对网格的引用时很常见，比如在粒子系统中。

4.3 材料

材料是一个重要的特征，决定了物体的绘制方式。虽然这是一个熟悉的功能，但如果使用不当，很容易造成内存泄漏。本节告诉你如何安全使用材料。

只通过访问参数来重复。

材料的主要注意事项是，仅仅通过访问它们的参数就可以复制它们。而很难注意到它正在被复制。

看一下下面的代码。

▼ 清单 4.3 被复制的材料的例子。

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

这是一个简单的过程，将材料的颜色属性设置为color.green。渲染器的材质被复制了。然后，复制的对象必须明确地被销毁。

▼ 清单 4.4 删除重复材料的例子

```
1、材料材质。  
2:  
3: void Awake()  
4: {  
5:     material = renderer.material;  
6:     Material.color = Color.green;  
7: }  
8:  
9: void OnDestroy()  
10: {  
11:     如果(材料 != null)  
12:     {  
13:         销毁(材料)  
14:     }  
15: }
```

以这种方式销毁重复的材料可以避免内存泄漏。

彻底清理产生的材料。

动态生成的材料也很容易出现内存泄漏。确保在你使用完后销毁已生成的材料。

请看下面的示例代码。

▼ 清单4.5。 动态生成的材料删除的例子

```

1、材料材质。
2:
3: void Awake()
4: {
5:     material = new Material(); // Material is generated dynamically.
6: }
7:
8: void OnDestroy()
9: {
10:    如果(材料 != null)
11:    {
12:        Destroy(material); // 销毁使用过的材料 13:    }
14: }
```

当你完成使用后, 销毁生成的材料(OnDestroy)。根据项目的规则和规范, 在适当的时间销毁材料。

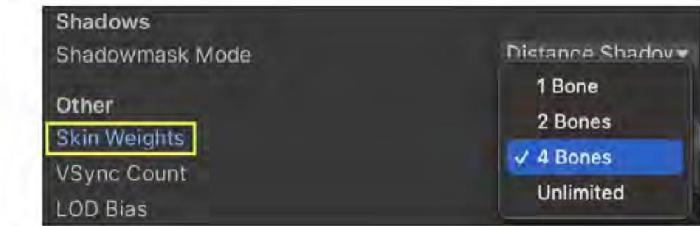
4.4 动画

动画是一种广泛使用的资产, 无论是2D还是3D。本节介绍了与Animation Clip和Animator有关的做法。

4.4.1 调整皮肤重量的数量

运动内部通过计算每个顶点受每个骨骼影响的程度来更新位置。在位置计算中考虑到的骨骼数量被称为皮肤重量或影响因素的数量。因此, 可以通过调整皮重物的数量来减少负荷。然而, 减少皮肤重量的数量可能会导致一个奇怪的外观, 所以在调整时要核实这一点。

皮肤权重的数量可以从'项目设置->质量'其他方面设置。



▲ 图4.8 调整皮肤重量

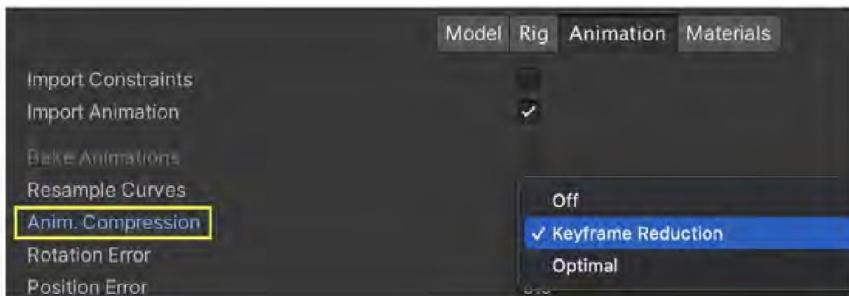
这个设置也可以从脚本中动态调整。因此，它可以被微调，例如，对于低规格的设备，皮肤权重设置为2，对于高规格的设备设置为4。

▼ 清单4. 6皮肤重量的配置变化

```
1: //如何完整地切换QualitySettings。  
2: // 参数编号按照配置屏幕上的Levels顺序从上往下依次为0、1...。 ... 按配置屏幕上的级别顺序排列。  
3: QualitySettings.SetQualityLevel(0);  
4:  
5: // 如何只改变SkinWeights  
6: QualitySettings.skinWeights = SkinWeights.TwoBones;
```

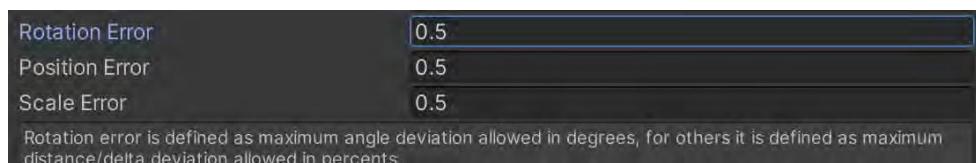
4.4.2 关键的减少

动画文件的存储和运行时的内存消耗很大，这取决于按键的数量。减少钥匙数量的一种方法是使用一种叫做Anim.Compression的功能。这个选项可以通过选择模型导入设置中的动画选项卡来找到；如果启用了Anim.Compression，在导入资产时就会自动删除不必要的键。



▲ 图4.9 Anim. 压缩设置屏幕。

如果数值的变化很小, 关键帧的减少会减少关键。具体来说, 如果与前一条曲线相比, 钥匙在误差(Error)范围内, 就会被删除。这个误差范围是可以调整的。



▲ 图4.10 错误设置。

让事情变得复杂一点的是, 误差设置中的每一项都有不同的数值单位--旋转是度, 位置和比例是百分比。捕获的图像的公差为旋转0.5度, 位置和比例为0.5%。详细的算法可以在Unity文档中找到*1。

最优是更令人困惑的, 但它比较了两种还原方法, 密集曲线格式和关键帧还原, 并采用了数据较小的那一种。要记住的关键点是, Dense的尺寸比Keyframe Reduction小。然而, 它往往比较嘈杂, 可能会导致动画质量下降。在了解了这些特点之后, 你应该目测一下实际的动画, 看看它是否可以接受。

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

4.4.3 减少了更新的频率

Animator默认更新每一帧，即使它不在屏幕上。有一个叫做“剔除模式”的选项，允许你改变这种更新方法。

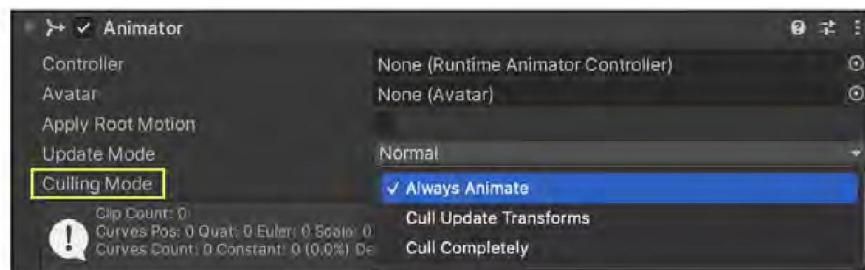


图4.11 剔除模式。

每个选项的含义如下。

▼ 表4.1 剔除模式的描述

类型。	意义
始终保持动画。	即使在屏幕外，也始终更新（默认设置）
Cull更新转换	离屏时不写IK或Transform。 状态机更新确实如此。
彻底清除	如果他们不在屏幕上，他们不会更新状态机。 动画完全停止。

关于这些选项，有一些要点需要注意。首先，在设置Cull Completely时，使用Root motion时必须注意。例如，如果你有一个动画从屏幕外的帧进来，该动画将立即停止，因为它是在屏幕外。其结果是，动画不会永远定格。

接下来是Cull Update Transform。这似乎是一个非常可用的选项，因为它只是跳过了转变的更新。然而，如果你有变形依赖的操作，如摇晃，就要小心。例如，如果角色定格了，更新将从那一刻的姿势跳过。当角色再次定格时，它将被更新为一个新的姿势，这可能会导致晃动的物体大幅移动。建议在改变设置之前，了解每个选项的利

弊。

这些设置也不允许对动态动画的更新频率进行微调。例如，你可以优化动画的更新频率，对于离摄像机较远的物体，将其减半。在这种情况下，你需要使用 AnimationClipPlayable 或者停用 Animator 并自己调用 Animator.Update。两者都需要编写自己的脚本，但后者比前者更容易实现。

4.5 粒子系统

游戏效果是游戏表现的一个重要部分，在 Unity 中经常使用粒子系统。本章从性能调整的角度介绍了粒子系统的使用，以及避免故障的注意点。

两个重要的观点是

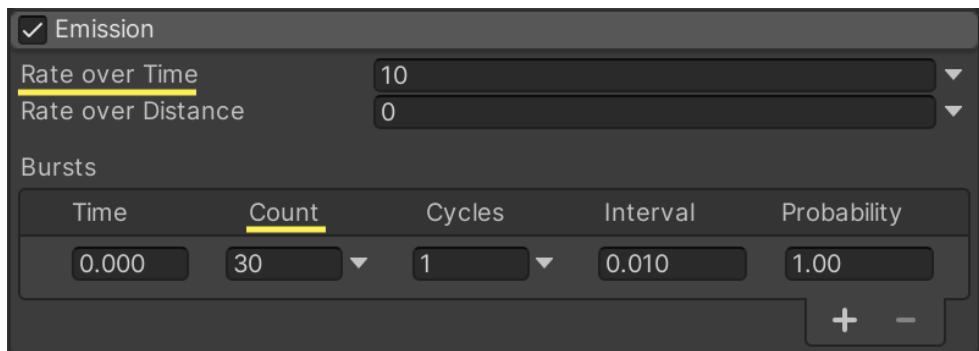
- 减少颗粒的数量。
- 注意，噪音很重。

4.5.1 减少颗粒的数量。

粒子的数量与负荷有关：由于粒子系统是由 CPU 驱动的（CPU 粒子），粒子越多，CPU 的负荷越高。作为一项基本政策，将颗粒的数量设置为必要的最小值。根据需要调整颗粒的数量。

有两种方法来限制粒子的数量。

- 排放的模块数量的限制。
- 主模块中的“最大颗粒”限制了释放的最大零件数量。



▲ 图4.12 排放模块对排放数量的限制

- 速率随时间变化：每秒钟释放的件数
- 突发事件 > 计数：在突发事件发生时要释放的件数

调整这些设置以达到所需的最小颗粒数。

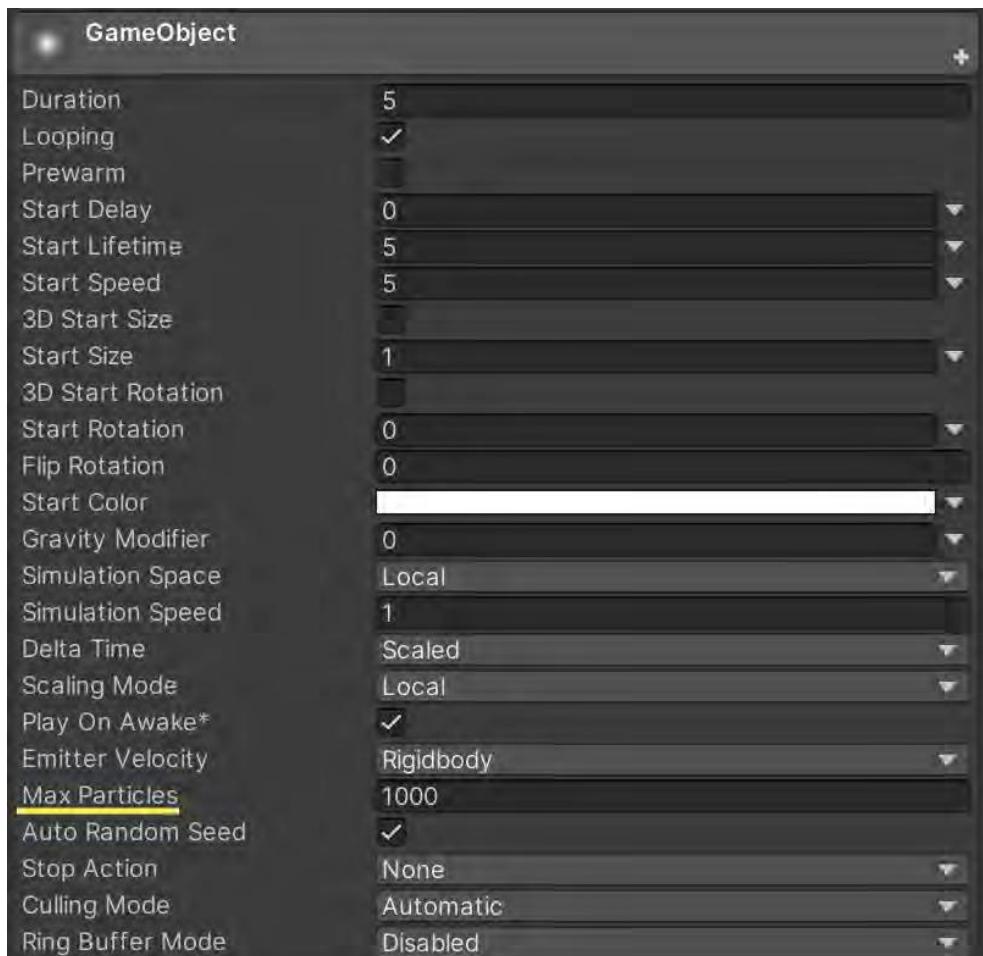


图4.13 最大颗粒限制了释放的零件数量。

另一种方法是主模块中的**Max Particles**。在上面的例子中，超过1000个颗粒将不再被排放。

还要注意副发射器。

在限制粒子的数量方面，子发射器模块也需要注意。

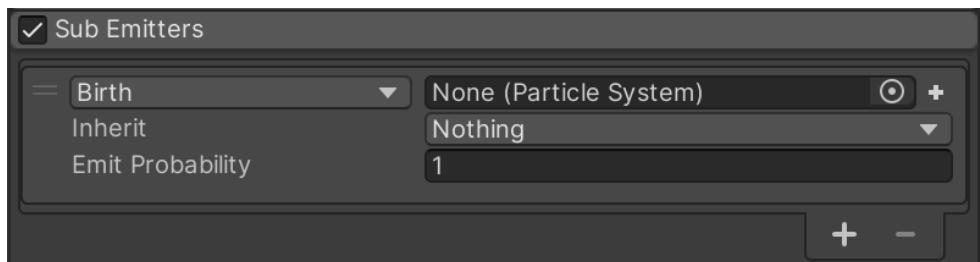


图4.14 子发射器模块。

子发射器模块在特定时间生成任意的粒子系统(例如，在创建时，在生命结束时，等等)，使用子发射器时要小心，因为有些设置可以一次性达到峰值数量。

4.5.2 注意，噪音很重。

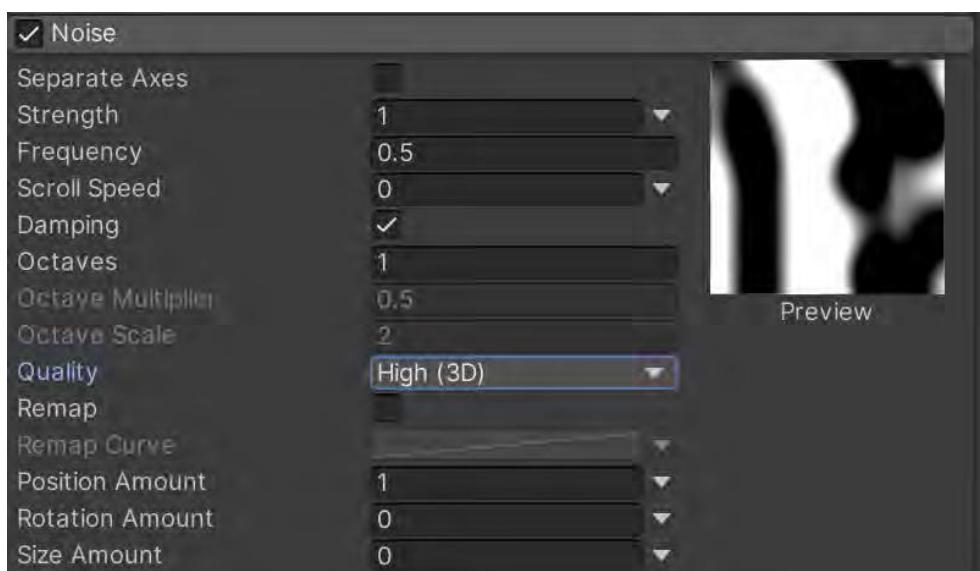


图4.15 噪声模块。

噪声模块的质量很容易过载。噪声可以用来创造有机颗粒，是提高效果质量的一个简单方法。

这种功能最常见的用途是在使用计算机的内存。因为它是一个经常使用的功能，你要小心它的性能。

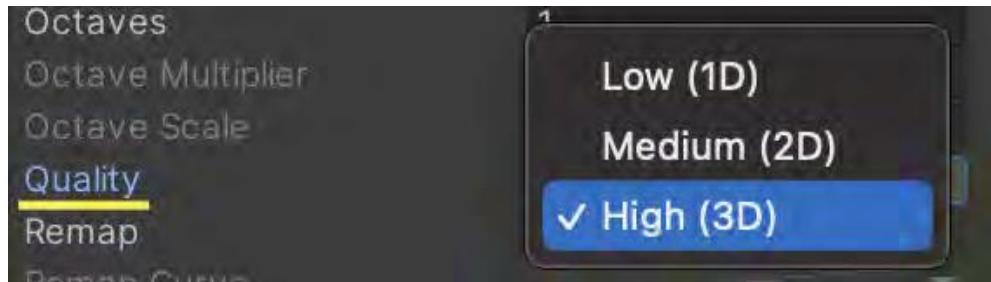


图4.16 噪声模块的质量

- 低 (1D)
- Midium (2D)
- 高 (3D)

质量的负荷随着尺寸的增加而增加。如果你不需要噪声，请关闭噪声模块。如果你需要使用噪音，先将质量设置为低，然后根据你的要求提高质量。

4.6 音频

导入声音文件的默认状态在性能上是一个改进点。有三个配置项目。

- 负载类型
- 压缩格式
- 强制转为单声道

将这些设置为适当的背景音乐、声音效果和游戏开发中常用的声音。

4.6.1 负载类型

有三种方法来加载声音文件(AudioClip)。



图4.17 AudioClip LoadType.

- 加载时解压
- 压缩在内存中
- 流媒体

加载时解压

加载时解压是将未压缩的内容加载到内存中；它对CPU的要求较低，因此播放时的等待时间较短。另一方面，它使用更多的内存。

建议用于长度较短、需要立即回放的声音效果；与背景音乐或长的语音文件一起使用时应注意，因为它们会占用大量的内存。

压缩在内存中

在内存中压缩，以压缩状态将AudioClip加载到内存中。这意味着它在播放时被解压缩。这意味着CPU负载很高，播放延迟很可能发生。

适用于文件大小较大，不希望直接部署到内存中的声音，或对有轻微播放延迟没有问题的声音。它经常与语音一起使用。

流媒体

流媒体，顾名思义，是一种加载和播放方法。内存使用率很低，但CPU负载很高。建议与长的BGM一起使用。

▼ 表4.2 装载方法和主要用途摘要

类型。	使用
-----	----

4.6 音频

加载时解压	音效
压缩在内存中	声音
流媒体	BGM