

声音的。	65
尽可能将声音剪辑设为单声道。	65
使用原始未压缩的 WAV 文件作为您的源资产。	67
压缩剪辑并降低压缩比特率。	67
选择正确的负载类型。	67
从内存中卸载静音的音频源。	67
动画片。	68
使用通用装备与人形装备	68
避免过度使用动画师	69
物理。	70
优化您的设置。	70
简化碰撞器。	72
使用物理方法移动刚体。	73
修复固定时间步长。	73
使用物理调试器进行可视化。	73
工作流程和协作。	74
使用版本控制。	74
分解大场景。	75
删除未使用的资源。	75
使用 Unity Accelerator 加速共享。	76
通过 Unity Integrated Success 消除障碍。	77
结论和后续步骤。	78

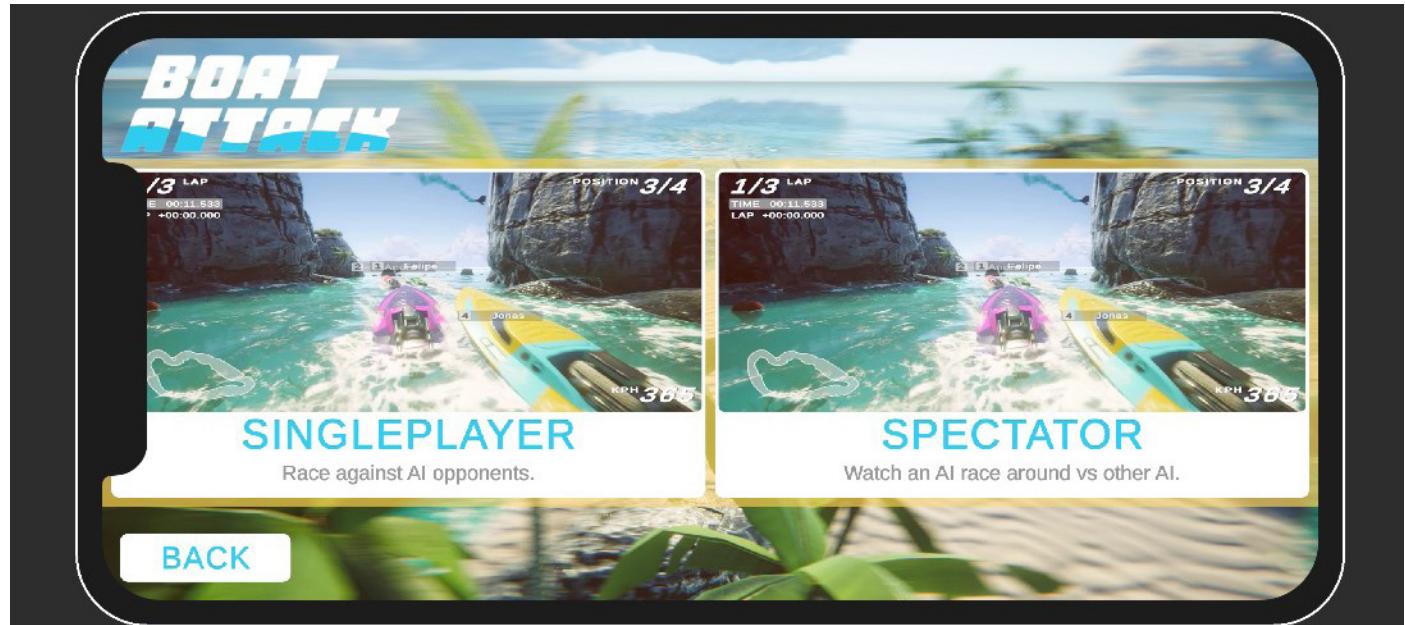
介绍

优化您的 iOS 和 Android 应用程序是支撑整个开发周期的重要过程。移动硬件不断发展，移动游戏的优化及其艺术、游戏设计、音频和货币化策略在塑造玩家体验方面发挥着关键作用。

iOS 和 Android 都拥有活跃的用户群数十亿。如果您的手机游戏经过高度优化，那么它通过特定平台商店认证的机会就更大。为了最大限度地提高您在发布时及以后获得成功的机会，您的目标始终是双重的：打造最流畅、最身临其境的体验，并使其在最广泛的手持设备上表现出色。

本指南汇集了 Unity 软件工程师专家团队的知识和建议。[Unity 的加速解决方案](#)游戏团队与整个行业的开发商合作，帮助推出最好的游戏。请按照此处概述的步骤操作，以获得移动游戏的最佳性能，同时降低其功耗。

在 Unity 团队的支持下开始优化。¹



请注意，此处讨论的许多优化可能会引入额外的复杂性，这可能意味着额外的维护和潜在的错误。在实施这些最佳实践时，平衡性能收益与时间和劳动力成本。

分析

尽早、经常在目标设备上进行配置

分析是衡量游戏运行时性能各方面的过程。通过使用分析工具，您可以测量游戏在目标平台上的运行情况，并使用此信息来追踪性能问题的原因。通过在进行更改时观察分析工具，您可以判断更改是否真正解决了性能问题。

这[统一分析器](#)提供有关您的应用程序的性能信息，但如果您不使用它，它无法为您提供帮助。

尽早并在整个开发周期中分析您的项目，而不仅仅是在接近交付时。一旦出现故障或峰值，就应立即调查，以对项目中发生重大更改之前和之后的性能进行基准测试。当您为项目开发“性能签名”时，您将能够更轻松地发现新问题。

虽然在编辑器中进行分析可以让您了解游戏中不同系统的相对性能，但在每个设备上进行分析可以让您有机会获得更准确的见解。尽可能分析目标设备上的开发构建。请记住针对您计划支持的最高和最低规格设备进行分析和优化。

与 Unity Profiler 一起，您可以利用[内存分析器](#)和[轮廓分析仪](#)以及来自 iOS 和 Android 的这些本机工具，用于在各自的硬件上进行进一步的性能测试：

— 在 iOS 上，使用[Xcode](#)和[仪器](#)。

— 在 Android/Arm 上使用：

[安卓工作室](#)：最新的 Android Studio 包含一个新的[Android 分析器](#)它取代了以前的 Android Monitor 工具。使用它来收集有关 Android 设备上硬件资源的实时数据。

[Arm移动工作室](#)：这套工具可以帮助您详细分析和调试游戏，适合运行 Arm 硬件的设备。

[Snapdragon 分析器](#)：仅适用于 Snapdragon 芯片组设备。分析 CPU、GPU、DSP、内存、电源、热量和网络数据，以帮助查找和修复性能瓶颈。

某些硬件也可以利用[英特尔VTune](#)，它可以帮助您查找并修复英特尔平台上的性能瓶颈（仅限英特尔处理器）。

看[分析使用 Unity 制作的应用程序](#)了解更多信息。

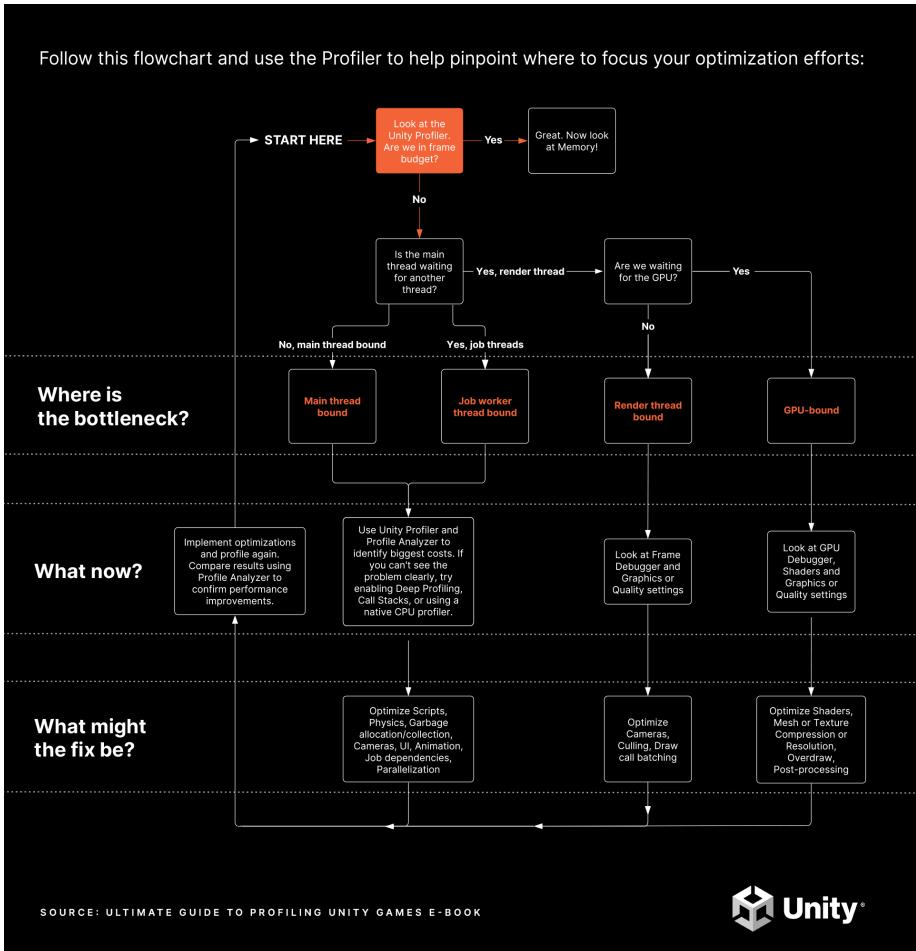
专注于优化正确的领域

不要猜测或假设是什么降低了游戏的性能。使用 Unity Profiler 和特定于平台的工具来定位延迟的精确来源。分析工具最终可以帮助您了解 Unity 项目的内部情况，但不要等到显着的性能问题开始出现才开始深入研究您的侦探工具箱。

当然，并非此处描述的每项优化都适用于您的应用程序。在一个项目中效果良好的东西可能无法转化为您的项目。找出真正的瓶颈，并将精力集中在对您的工作有利的事情上。

要了解有关如何规划分析工作流程的更多信息，请参阅[Unity 游戏分析终极指南](#)。





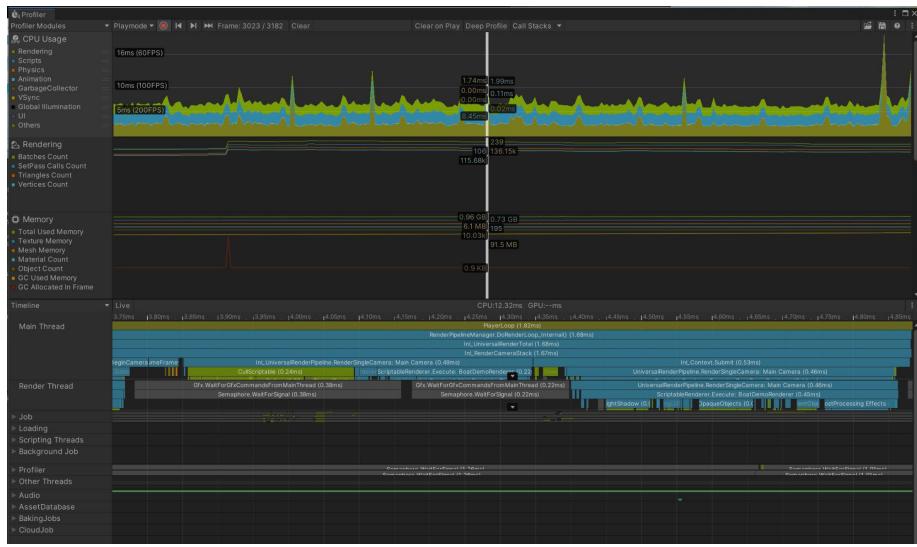
分析电子书中的图表，其中包含您可以遵循的工作流程，以有效地分析您的 Unity 项目。

了解 Unity Profiler 的工作原理

内置的[统一分析器](#)可以帮助您检测运行时任何滞后或冻结的原因，并更好地了解特定帧或时间点发生的情况。

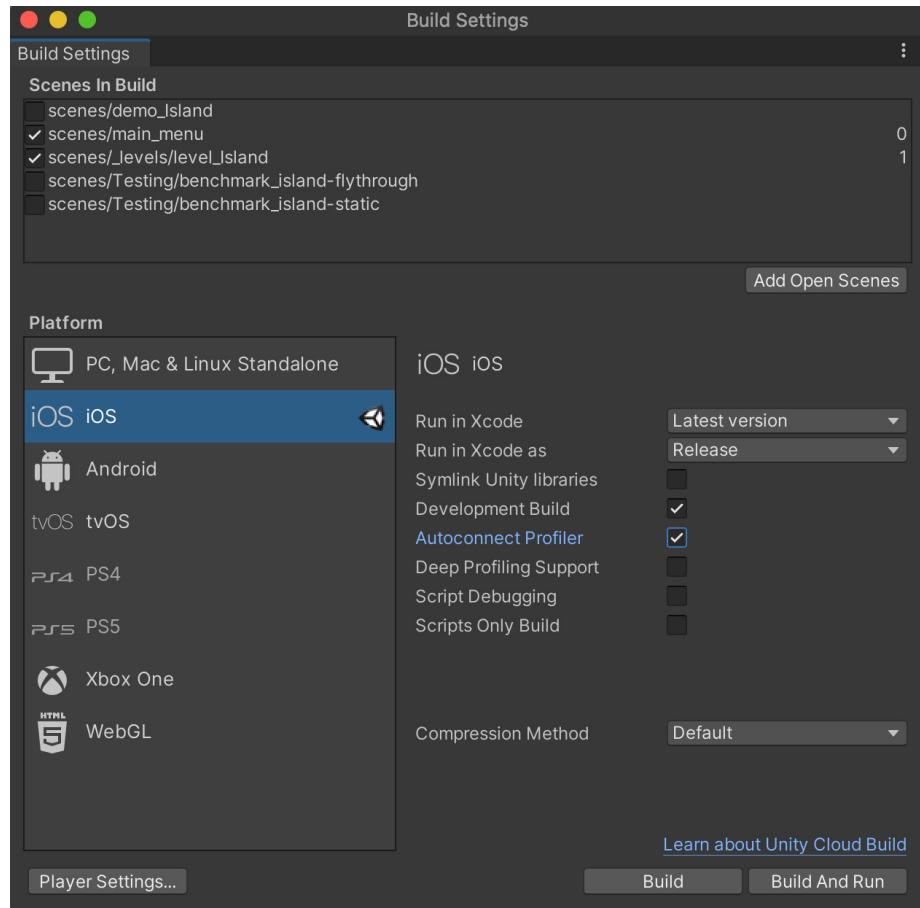
Profiler 是基于仪器的；它分析自动标记的游戏和引擎代码的计时（例如 MonoBehaviour 的 Start 或 Update 方法，或特定的 API 调用），或在[分析器标记器API](#)。

首先启用 CPU 和内存轨道作为默认设置。您可以根据游戏需要（例如，物理重或基于音乐的游戏玩法）监控补充分析器模块，例如渲染器、音频和物理。



使用 Unity Profiler 测试应用程序的性能和资源分配。

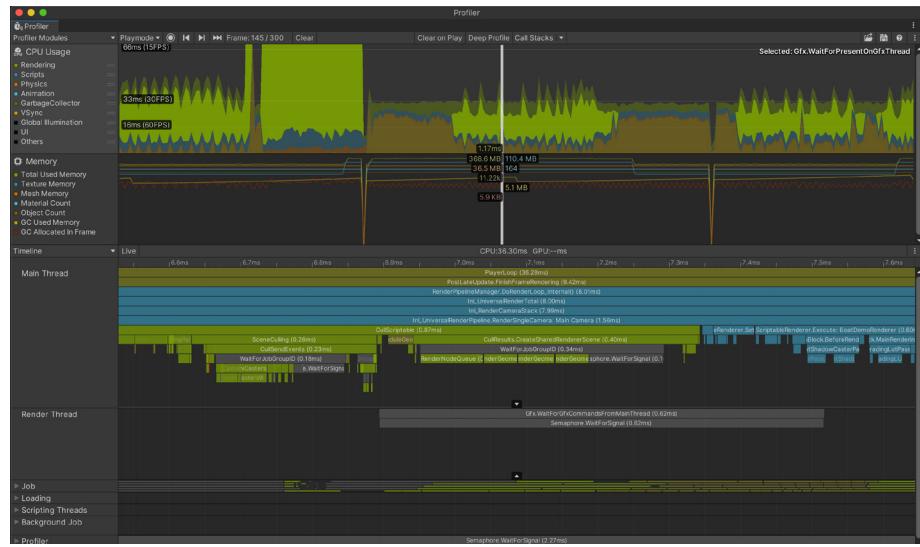
要从所选平台内的实际移动设备捕获分析数据，请在单击“构建并运行”之前选中“开发构建”和“自动连接分析器”框。或者，如果您希望应用程序与分析程序分开发起，您可以取消选中“自动连接分析器”框，然后在应用程序运行后手动连接。



在分析之前调整构建设置。

选择要分析的平台目标。“录制”按钮可跟踪应用程序播放的几秒钟（默认情况下为 300 帧）。

如果您需要更长的捕获时间，请转至 **Unity > Preferences > Analysis > Profiler > Frame Count** 将其增加至 2000。虽然这意味着 Unity 编辑器必须执行更多的 CPU 工作并占用更多的内存，但它可能会很有用，具体取决于您的具体场景。

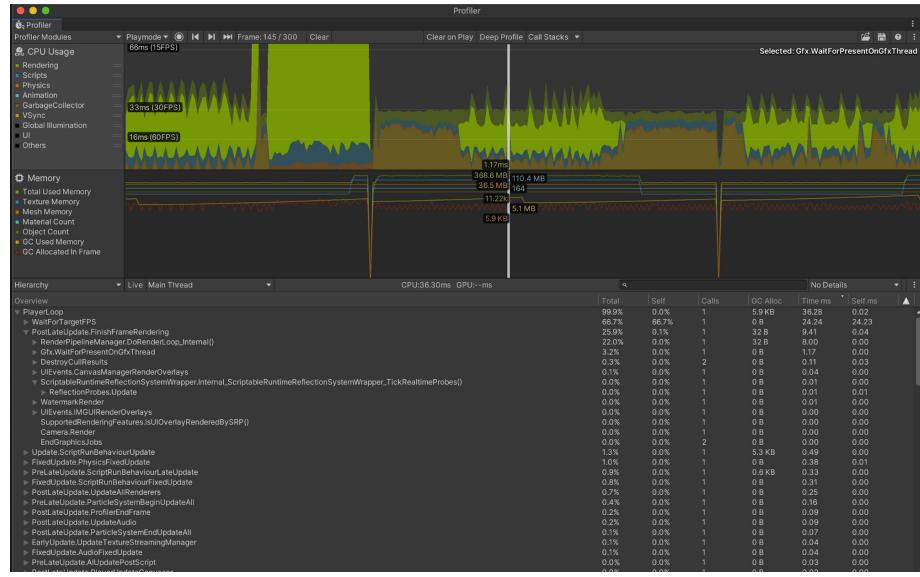


使用时间轴视图来确定您是受 CPU 限制还是 GPU 限制。

当使用**深度剖析**设置中，Unity 可以分析脚本代码中每个函数调用的开始和结束，准确告诉您应用程序的哪一部分正在执行并可能导致延迟。然而，深度分析会增加每个方法调用的开销，并且可能会影响性能分析。

在窗口中单击以分析特定帧。接下来，使用时间轴或层次结构视图进行以下操作：

- 时间轴显示特定帧的时间的视觉细分。这使您可以直观地看到活动如何相互关联以及跨不同线程的关联。使用此选项可以确定您是否受 CPU 限制或 GPU 限制。
- 层次结构显示了分组在一起的 ProfileMarkers 的层次结构。这允许您根据以毫秒为单位的时间成本（Time ms 和 Self ms）对样本进行排序。您还可以统计帧上函数的调用次数和托管堆内存（GC Alloc）。



层次结构视图允许您按时间成本对 ProfileMarkers 进行排序。

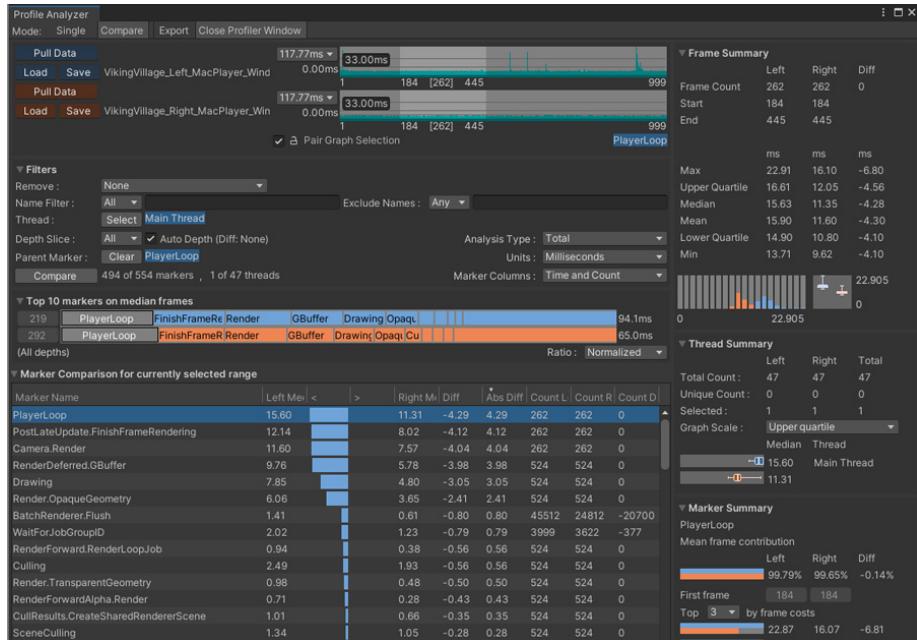
您可以找到 Unity Profiler 的完整概述[这里](#)。如果您是分析新手，也可以观看此内容

Unity 分析简介

在优化项目中的任何内容之前，请保存 Profiler .data 文件。实施您的更改并比较修改前后保存的数据。依靠这个循环来提高性能：分析、优化和比较。然后，冲洗并重复。

使用配置文件分析器

Profile Analyzer 允许您聚合多个 Profiler 数据帧，然后定位感兴趣的帧。您想看看对项目进行更改后 Profiler 会发生什么情况吗？比较视图允许您加载和区分两个数据集，以便您可以测试更改并改进其结果。这[轮廓分析仪](#)可通过 Unity 的包管理器获得。



使用以下工具更深入地了解帧和标记数据 [轮廓分析仪](#)，它补充了现有的 Profiler。

处理每帧的特定时间预算

每个帧都有一个基于每秒目标帧数 (fps) 的时间预算。对于以 30 fps 运行的应用程序，其帧预算不能超过每帧 33.33 毫秒 (1000 ms/30 fps)。同样，60 fps 的目标为每帧留下 16.66 毫秒。

考虑设备温度

但是，对于移动设备，我们不建议始终使用此最大时间，因为设备可能会过热，并且操作系统可能会对 CPU 和 GPU 进行热节流。我们建议您仅使用大约 65% 的可用时间来进行帧之间的冷却。典型的帧预算约为 30 fps 时每帧 22 毫秒，60 fps 时每帧约 11 毫秒。

设备可能会在短时间内超出此预算（例如，对于过场动画或加载序列），但不会长时间超出此预算。

大多数移动设备不像桌面设备那样具有主动冷却功能。物理热量水平会直接影响性能。

如果设备运行过热，Profiler 可能会感知并报告性能不佳，即使这不会引起长期关注。为了防止分析过热，应短时间进行分析。这可以冷却设备并模拟真实世界的条件。我们的一般建议是让设备冷却 10-15 分钟，然后再次进行分析。

确定您是 GPU 密集型还是 CPU 密集型

中央处理单元（CPU）负责确定必须绘制什么，图形处理单元（GPU）负责绘制它。当渲染性能问题是由于 CPU 渲染一帧的时间过长而导致时，游戏就会成为 CPU 密集型游戏。当渲染性能问题是由于 GPU 渲染帧的时间过长而导致时，就会出现 GPU 限制。

Profiler 可以告诉您 CPU 花费的时间是否超过了分配的帧预算，或者罪魁祸首是否是您的 GPU。它通过发出前缀为 Gfx 的标记来实现此目的，如下所示：

- 如果您看到 Gfx.WaitForCommands 标记，则渲染线程已准备就绪，但您可能正在等待主线程上的瓶颈。
- 如果您经常遇到 Gfx.WaitForPresent，这意味着主线程已准备好，但正在等待 GPU 呈现帧。

在最低规格设备上进行测试

有各种各样的 iOS 和 Android 设备。

我们想重申尽可能根据您希望应用程序支持的最小和最大设备规格测试您的项目的重要性。

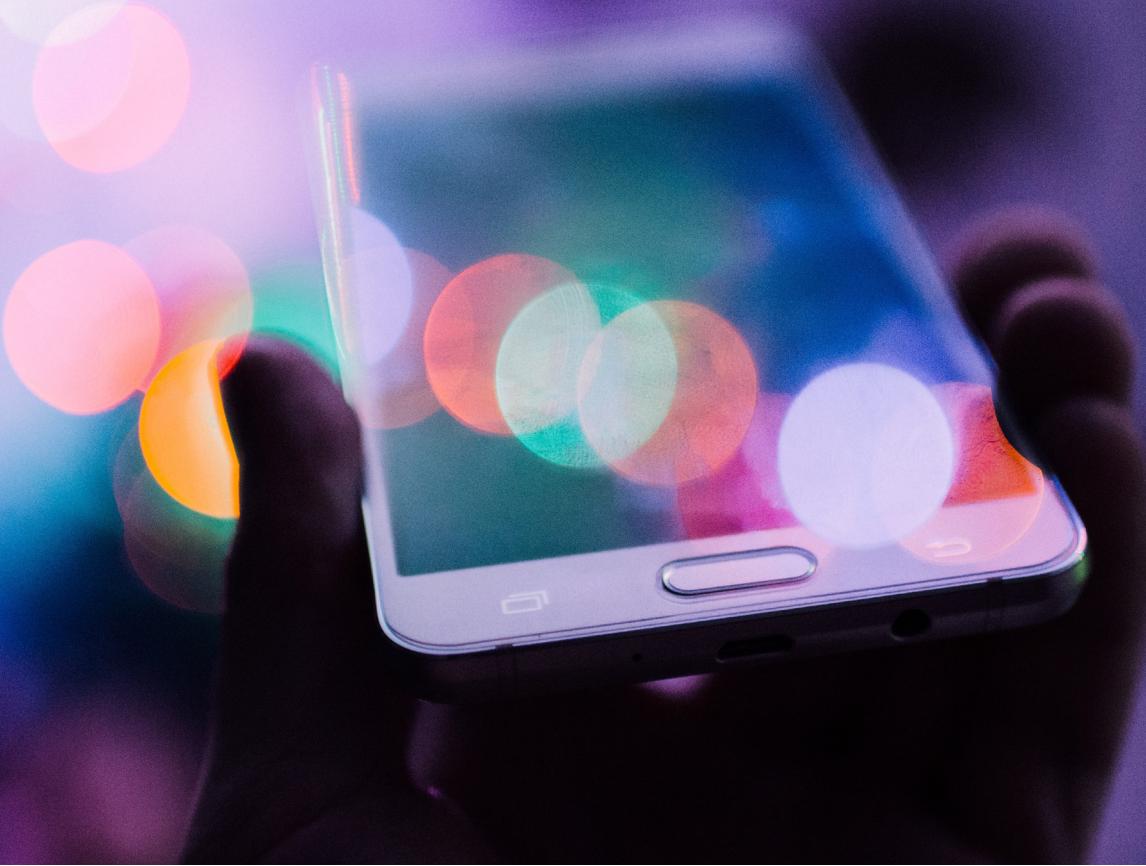
记忆

Unity 对用户生成的代码和脚本采用自动内存管理。小块数据，如值类型局部变量，分配在堆栈上。较大的数据块和长期存储被分配到托管堆或本机堆。

垃圾收集器定期识别并释放未使用的托管堆内存。资产垃圾收集根据需要或在加载新场景并释放本机对象和资源时运行。虽然它会自动运行，但检查堆中所有对象的过程可能会导致游戏卡顿或运行缓慢。

优化内存使用意味着要注意何时分配和释放堆内存，以及如何最大限度地减少垃圾收集的影响。

看[了解托管堆](#)了解更多信息。

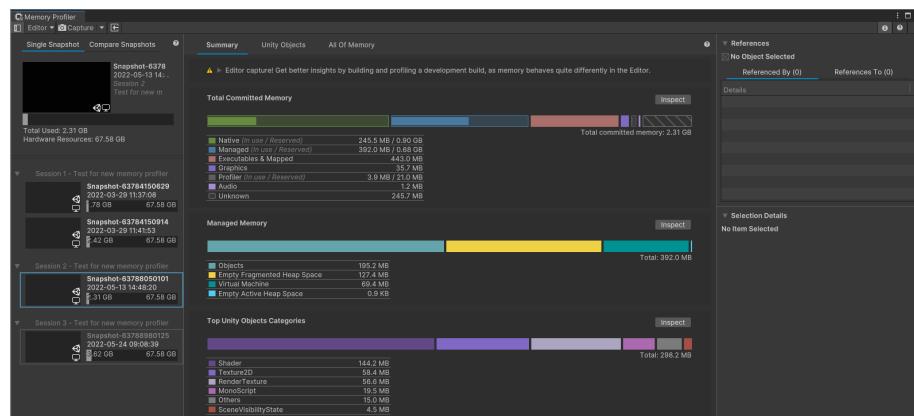


使用内存分析器

这[内存分析器包](#)拍摄托管堆内存的快照，以帮助您识别碎片和内存泄漏等问题。

使用 Unity 对象选项卡来确定可以消除重复内存条目的区域或查找哪些对象使用最多内存。All of Memory 选项卡显示 Unity 跟踪的快照中所有内存的详细信息。

了解如何利用[Unity 中的内存分析器](#)以提高内存使用率。



在内存分析器中捕获、检查和比较快照。

减少垃圾收集 (GC) 的影响

统一使用[Boehm-Demers-Weiser 垃圾收集器](#)，它会停止运行您的程序代码，并且只有在其工作完成后才恢复正常执行。

请注意某些不必要的堆分配，这可能会导致 GC 峰值：

- 字符串：在 C# 中，字符串是引用类型，而不是值类型。
减少不必要的字符串创建或操作。避免解析基于字符串的数据文件，例如 JSON 和 XML；以 ScriptableObjects 或 MessagePack 或 Protobuf 等格式存储数据。使用[字符串生成器](#)如果您需要在运行时构建字符串，则使用 .class。
- Unity 函数调用：请注意，某些函数会创建堆分配。缓存对数组的引用，而不是在循环中间分配它们。另外，利用某些避免产生垃圾的功能；例如，使用 GameObject .CompareTag 而不是手动将字符串与 GameObject .tag 进行比较（返回新字符串会产生垃圾）。
- 装箱：避免传递值类型变量来代替引用类型变量。这会创建一个临时对象，并且随之而来的潜在垃圾（例如，int i = 123; object o = i）将值类型隐式转换为类型 object。相反，尝试使用您想要传入的值类型提供具体的覆盖。泛型也可用于这些覆盖。
- 协程：虽然yield不会产生垃圾，但创建一个新的 WaitForSeconds对象会产生垃圾。缓存并重用 WaitForSeconds 对象，而不是在屈服线中创建它。
- LINQ 和正则表达式：这两者都会从幕后拳击中生成垃圾。如果性能是一个问题，请避免使用 LINQ 和正则表达式。

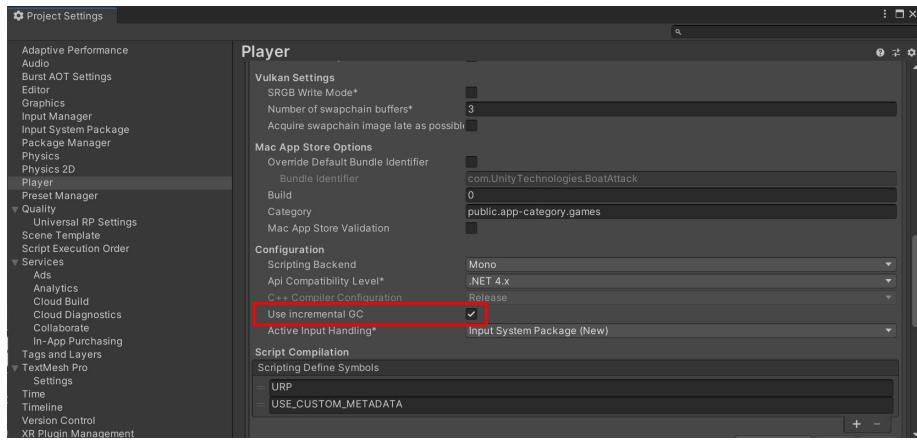
有关更多信息，请参阅手册页[垃圾收集最佳实践](#)。

如果可能的话，定时垃圾收集

如果您确定垃圾收集冻结不会影响游戏中的特定点，则可以使用 System .GC .Collect 触发垃圾收集。

看[了解自动内存管理](#)有关如何利用这一点为您带来优势的示例。²

²请注意，使用 GC 可能会给某些 C# 调用添加读写障碍，这些调用的开销很小，每帧脚本调用开销最多可能增加约 1 毫秒。为了获得最佳性能，理想的做法是在主游戏循环中没有 GC 分配，并将 GC 隐藏在用户不会注意到的地方。



使用增量垃圾收集器来减少 GC 峰值。

使用增量垃圾收集器来分割GC工作负载

增量垃圾收集不是使用程序执行的单个长时间中断，而是使用多个更短的中断，将工作负载分配到许多帧上。如果垃圾收集影响性能，请尝试启用此选项，看看它是否可以显着减少 GC 峰值问题。使用配置文件分析器来验证对您的应用程序的好处。

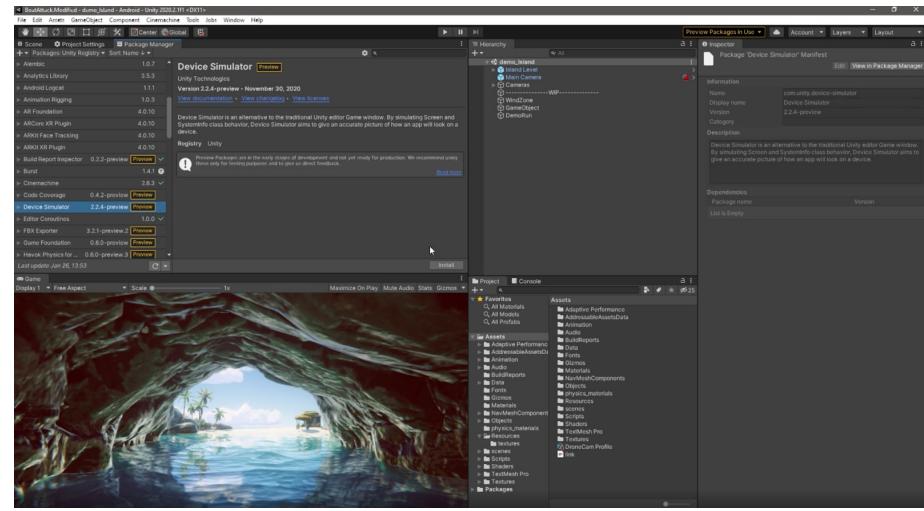
自适应表现

借助 Unity 和 Samsung**自适应性能**，您可以监控设备的热量和电源状态，以确保您准备好做出适当的反应。当用户长时间玩游戏时，您可以动态降低细节级别 (LOD) 偏差，以帮助您的游戏继续平稳运行。自适应性能允许开发人员以受控方式提高性能，同时保持图形保真度。

虽然您可以使用自适应性能 API 来微调您的应用程序，但此软件包还提供自动模式。在这些模式下，自适应性能根据几个关键指标确定游戏设置：

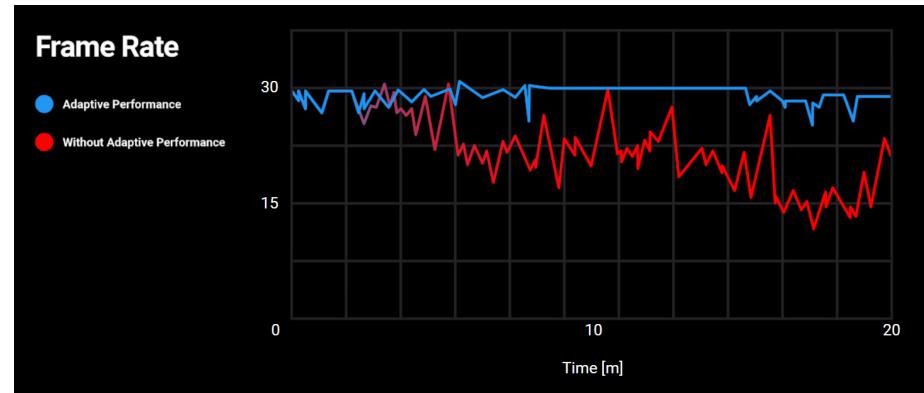
- 基于先前帧的所需帧速率
- 设备温度等级
- 设备接近热事件
- 受 CPU 或 GPU 绑定的设备

这四个指标决定了设备的状态，自适应性能会调整调整后的设置以减少瓶颈。这是通过提供一个整数值（称为索引器）来描述设备的状态来完成的。

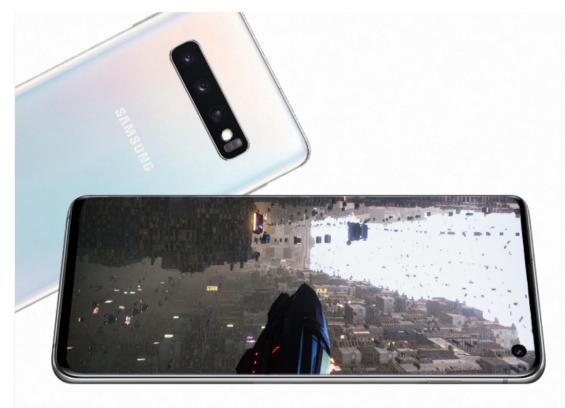


请注意，自适应性能仅适用于三星设备。

要了解有关自适应性能的更多信息，您可以查看[样品](#)我们通过选择 Package Manager > Adaptive Performance > Samples 在 Package Manager 中提供。每个示例都与特定的缩放器交互，因此您可以看到不同的缩放器如何影响您的游戏。我们还建议您查看[最终用户文档](#)了解有关自适应性能配置以及如何直接与 API 交互的更多信息。



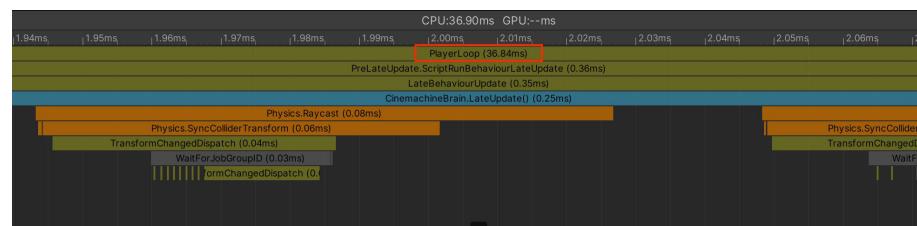
仅适用于三星设备。



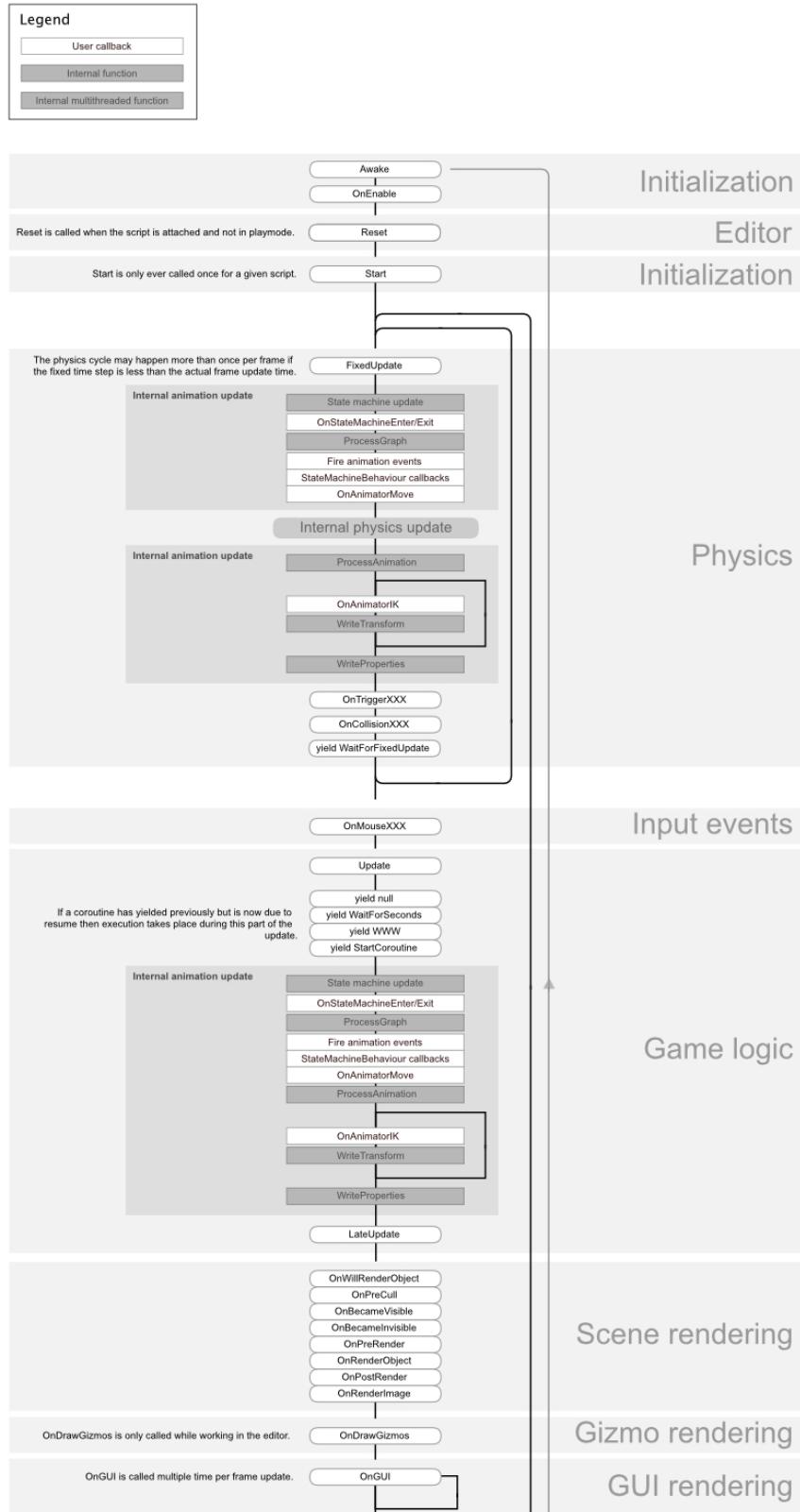
编程和代码架构

团结[玩家循环](#)包含与游戏引擎核心交互的函数。这种树状结构包括许多处理初始化和每帧更新的系统。您的所有脚本都将依赖此 PlayerLoop 来创建游戏玩法。

进行分析时，您将在 PlayerLoop 下看到项目的所有用户代码（编辑器组件位于 EditorLoop 下）。



Profiler 将在整个引擎执行的上下文中显示您的自定义脚本、设置和图形。



了解 PlayerLoop 和[脚本的生命周期](#)。

您可以使用这些提示和技巧来优化您的脚本。

了解 Unity PlayerLoop

确保您了解[执行顺序](#)Unity 的帧循环。每个 Unity 脚本都按预定顺序运行多个事件函数。您应该了解 Awake、Start、Update 和其他创建脚本生命周期的函数之间的区别。您可以利用低级 API 将自定义逻辑添加到播放器的更新循环中。

请参阅[脚本生命周期流程图](#)对于事件函数的具体执行顺序。

最小化每帧运行的代码

考虑代码是否必须在每一帧运行。将不必要的逻辑从 Update、LateUpdate 和 FixedUpdate 中移出。这些事件函数是放置必须更新每一帧的代码的方便位置，但提取不需要以该频率更新的任何逻辑。只要有可能，只在事情发生变化时执行逻辑。

如果你做需要使用 Update，请考虑每隔一段时间运行一次代码 n 帧。这是应用时间切片的一种方法，时间切片是一种在多个帧之间分配繁重工作负载的常用技术。在此示例中，我们每三帧运行一次 ExampleExpressiveFunction：

```
私有整数间隔 = 3;

无效更新 () {
{
    if (Time.frameCount % 间隔 == 0) {

        示例昂贵函数 () ;
    }
}
```

更好的是，如果 ExampleExpensiveFunction 对一组数据执行某些操作，请考虑使用时间切片来每帧对该数据的不同子集进行操作。通过做 $1/n$ 每一帧的工作而不是每一帧的全部工作 n 帧，您最终会获得整体上更加稳定和可预测的性能，而不是看到周期性的 CPU 峰值。

避免启动/唤醒中的繁重逻辑

当您的第一个场景加载时，将为每个对象调用这些函数：

- 苏醒
- 开启
- 开始

在应用程序呈现其第一帧之前，请避免在这些函数中使用昂贵的逻辑。否则，您可能会遇到比所需时间更长的加载时间。

请参阅[事件函数的执行顺序](#)有关第一个场景加载的详细信息。

避免空的 Unity 事件

即使是空的 MonoBehaviours 也需要资源，因此您应该删除空白的 Update 或 LateUpdate 方法。

如果您使用这些方法进行测试，请使用预处理器指令：

```
# 如果 UNITY_EDITOR  
无效更新 ()  
{  
}  
# 万一
```

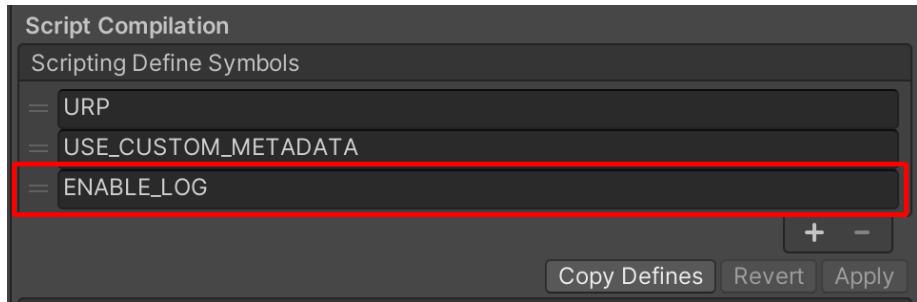
在这里，您可以自由地使用编辑器中的更新进行测试，而不会给您的构建带来不必要的开销。

删除调试日志语句

日志语句（尤其是在 Update、LateUpdate 或 FixedUpdate 中）可能会降低性能。在进行构建之前禁用 Log 语句。

为了更容易地做到这一点，请考虑制作一个[条件属性](#)以及预处理指令。例如，创建一个如下所示的自定义类：

```
公共静态类日志记录{  
  
    [System.Diagnostics.Conditional( "ENABLE_LOG" )] static  
    public void Log(对象消息)  
    {  
        UnityEngine.Debug.Log(消息);  
    }  
}
```



添加自定义预处理器指令可以让您对脚本进行分区。

使用您的自定义类生成日志消息。如果您在播放器设置中禁用 `ENABLE_LOG` 预处理器，所有 Log 语句都会一举消失。

同样的事情也适用于 `Debug` 类的其他用例，例如 `Debug.DrawLine` 和调试 `.DrawRay`。它们也只适合在开发过程中使用，并且会显著影响性能。

使用哈希值代替字符串参数

Unity 不使用字符串名称在内部寻址动画器、材质和着色器属性。为了速度，所有属性名称都被散列成属性 ID，并且这些 ID 实际上用于对属性进行寻址。

在动画师、材质或着色器上使用 `Set` 或 `Get` 方法时，请利用整数值方法而不是字符串值方法。字符串方法只是执行字符串散列，然后将散列 ID 转发到整数值方法。

使用[动画师.StringToHash](#)对于 `Animator` 属性名称和
[着色器.PropertyToID](#)对于材质和着色器属性名称。在初始化期间获取这些哈希值，并将它们缓存在变量中，以便在需要时将它们传递给 `Get` 或 `Set` 方法。

选择正确的数据结构

当您每帧迭代数千次时，您选择的数据结构可能会对效率或低效产生累积影响。对集合使用列表、数组或字典是否更有意义？跟着 [MSDN 数据结构指南](#)C# 中作为选择正确结构的一般指南。

避免在运行时添加组件

在运行时调用 `AddComponent` 会带来一些成本。每当在运行时添加组件时，Unity 都必须检查重复的组件或其他所需的组件。

[实例化预制件](#) 已设置所需的组件通常性能更高。

缓存游戏对象和组件

`GameObject.Find`、`GameObject.GetComponent` 和 `Camera.main`（在 2020.2 之前的版本中）可能会很昂贵，因此请避免在 `Update` 方法中调用它们。相反，在 `Start` 中调用它们并缓存结果。

例如，这表明重复 `GetComponent` 调用的使用效率低下：

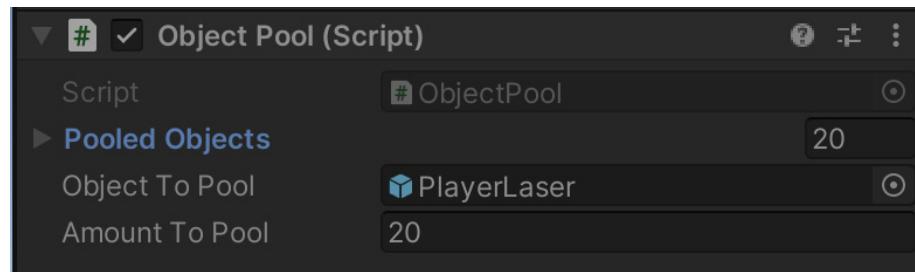
```
无效更新 ()  
{  
    渲染器 myRenderer = GetComponent<渲染器>(); 示例函数  
    (myRenderer) ;  
}
```

仅调用一次 `GetComponent` 会更有效，因为该函数的结果会被缓存。缓存的结果可以在 `Update` 中重用，而无需进一步调用 `GetComponent`。

```
私有渲染器 myRenderer; 无效开始 ()  
  
{  
    myRenderer = GetComponent<渲染器>();  
}  
  
无效更新 ()  
{  
    示例函数 (myRenderer) ;  
}
```

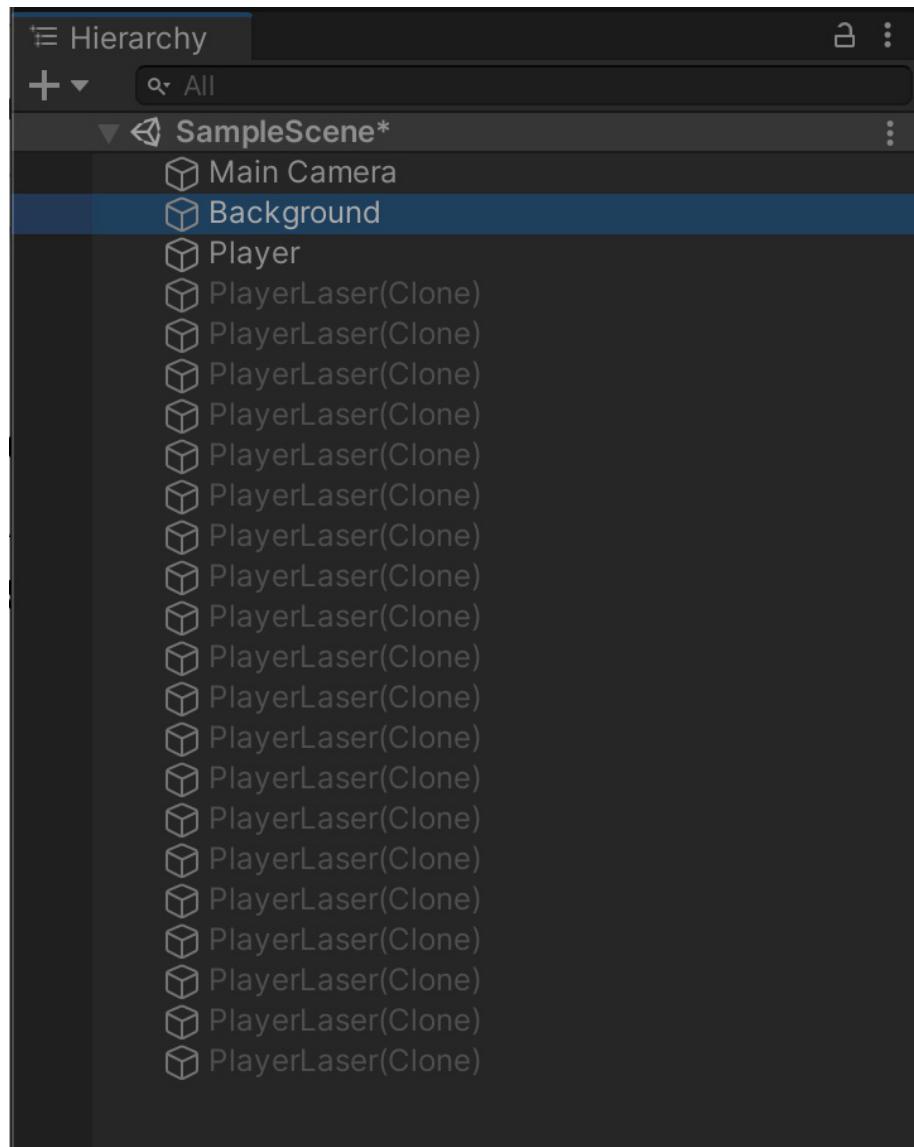
使用对象池

实例化和销毁会产生垃圾和垃圾收集 (GC) 峰值，并且通常是一个缓慢的过程。



在此示例中，`ObjectPool` 创建 20 个 `PlayerLaser` 实例以供重用。

当 CPU 峰值不太明显时，在游戏中的某个点（例如，在菜单屏幕期间）创建可重用实例。使用集合来跟踪这个对象“池”。在游戏过程中，在需要时启用下一个可用实例，禁用对象而不是销毁它们，并将它们返回到池中。



PlayerLaser 对象池处于非活动状态并准备射击。

这减少了项目中托管分配的数量，并可以防止垃圾收集问题。

了解如何在 Unity 中创建简单的对象池系统[这里](#)。您还可以看到[这些代码示例](#)，来自 Unity，使用对象池来指导您自己的游戏开发。

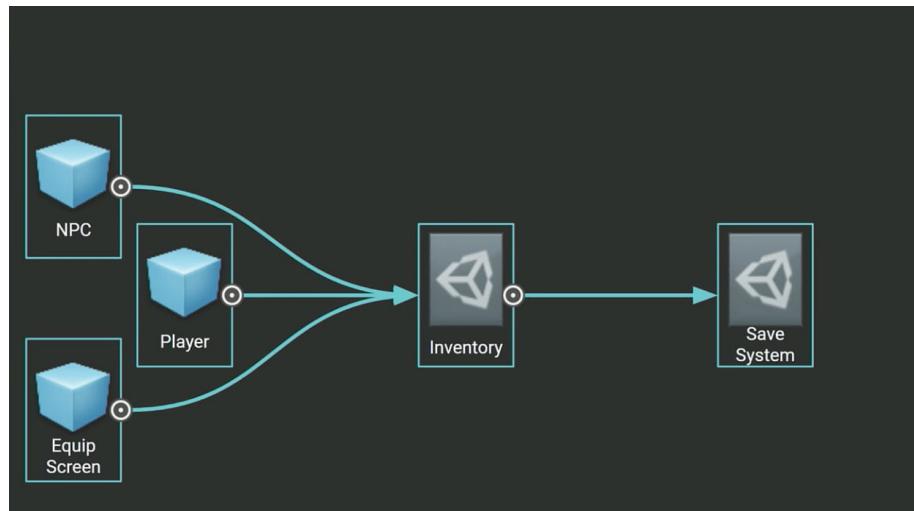
使用脚本化对象

将不变的值或设置存储在 ScriptableObject 而不是 MonoBehaviour 中。ScriptableObject 是一种存在于项目内部的资产，您只需设置一次。它不能直接附加到 GameObject。

MonoBehaviours 会带来额外的开销，因为它们需要一个 GameObject（默认情况下需要一个 Transform）来充当主机。这意味着您需要在存储单个值之前创建大量未使用的数据。ScriptableObject 通过删除 GameObject 和 Transform 来减少内存占用。它还在项目级别存储数据，如果您需要从多个场景访问相同的数据，这会很有帮助。

一个常见的用例是有许多游戏对象依赖于不需要在运行时更改的相同重复数据。您可以将其汇集到 ScriptableObject 中，而不是在每个 GameObject 上都有重复的本地数据。然后，每个对象存储对共享数据资产的引用，而不是复制数据本身。这可以显着提高具有数千个对象的项目的性能。

在 ScriptableObject 中创建字段来存储您的值或设置，然后在 MonoBehaviours 中引用 ScriptableObject。



在此示例中，名为 Inventory 的 ScriptableObject 保存各种 GameObject 的设置。

每次使用该 MonoBehaviour 实例化对象时，使用 ScriptableObject 中的这些字段可以防止不必要的数据重复。

在软件设计中，这是一种称为优化[蝇量模式](#)。使用 ScriptableObjects 以这种方式重构代码可以避免复制大量值并减少内存占用。

看这个[ScriptableObject 简介](#)devlog 了解 ScriptableObjects 如何使您的项目受益。您还可以找到相关文档[这里](#)或我们的技术指南[使用 ScriptableObjects 在 Unity 中创建模块化游戏架构](#)。

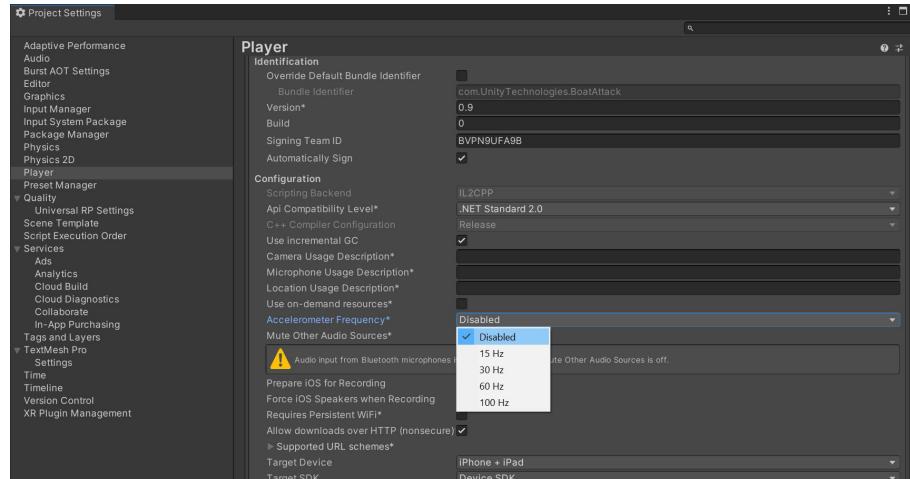
要了解有关在 Unity 中使用设计模式的更多信息，请参阅电子书[使用游戏编程模式升级您的代码](#)了解有关使用设计模式的更多信息。

项目配置

有一些项目设置可能会影响您的移动性能。

降低或禁用加速度计频率

Unity 每秒会多次汇集您移动设备的加速计。如果您的应用程序中未使用它，请禁用它，或者降低其频率以获得更好的性能。



如果您不在手机游戏中使用加速计频率，请确保禁用它。

禁用不必要的播放器或质量设置

在播放器设置中，针对不受支持的平台禁用 Auto Graphics API，以防止生成过多的着色器变体。如果您的应用程序不支持较旧的 CPU，请禁用它们的目标架构。

在质量设置中，禁用不需要的质量级别。

禁用不必要的物理

如果您的游戏不使用物理，请取消选中“自动模拟”和“自动同步变换”。这些只会减慢您的应用程序，而没有明显的好处。

选择正确的帧速率

移动项目必须在帧速率与电池寿命和热节流之间取得平衡。不要以 60 fps 的速度突破设备的限制，而是考虑以 30 fps 的速度运行作为折衷方案。Unity 对于移动设备默认为 30 fps。

您还可以使用 `Application.targetFrameRate` 在运行时动态调整帧速率。例如，对于慢速或相对静态的场景，您甚至可以将帧速率降至 30 fps 以下，并为游戏玩法保留更高的 fps 设置。

避免大的层次结构

分裂你的等级制度。如果您的游戏对象不需要嵌套在层次结构中，请简化父子关系。较小的层次结构受益于多线程来刷新场景中的变换。复杂的层次结构会产生不必要的转换计算以及更多的垃圾收集成本。

看“[优化层次结构](#)”在 Unity 博客上和这个[团结谈话](#)了解 `Transforms` 的最佳实践。

转变一次，而不是两次

另外，移动变换时，请使用[变换.SetPositionAndRotation](#) 立即更新位置和旋转。这避免了两次修改变换的开销。

如果你需要[实例化](#)运行时的 `GameObject`，一个简单的优化是在实例化期间进行父级和重新定位：

```
GameObject.Instantiate(预制件, 父级); GameObject .Instantiate(预  
制件、父级、位置、旋转);
```

有关 `Object.Instantiate` 的更多详细信息，请参阅[脚本 API](#)。

假设垂直同步已启用

移动平台不会渲染半帧。即使您在编辑器中禁用垂直同步（项目设置 > 质量），垂直同步也会在硬件级别启用。如果 GPU 刷新速度不够快，当前帧将被保留，从而有效降低 fps。

资产

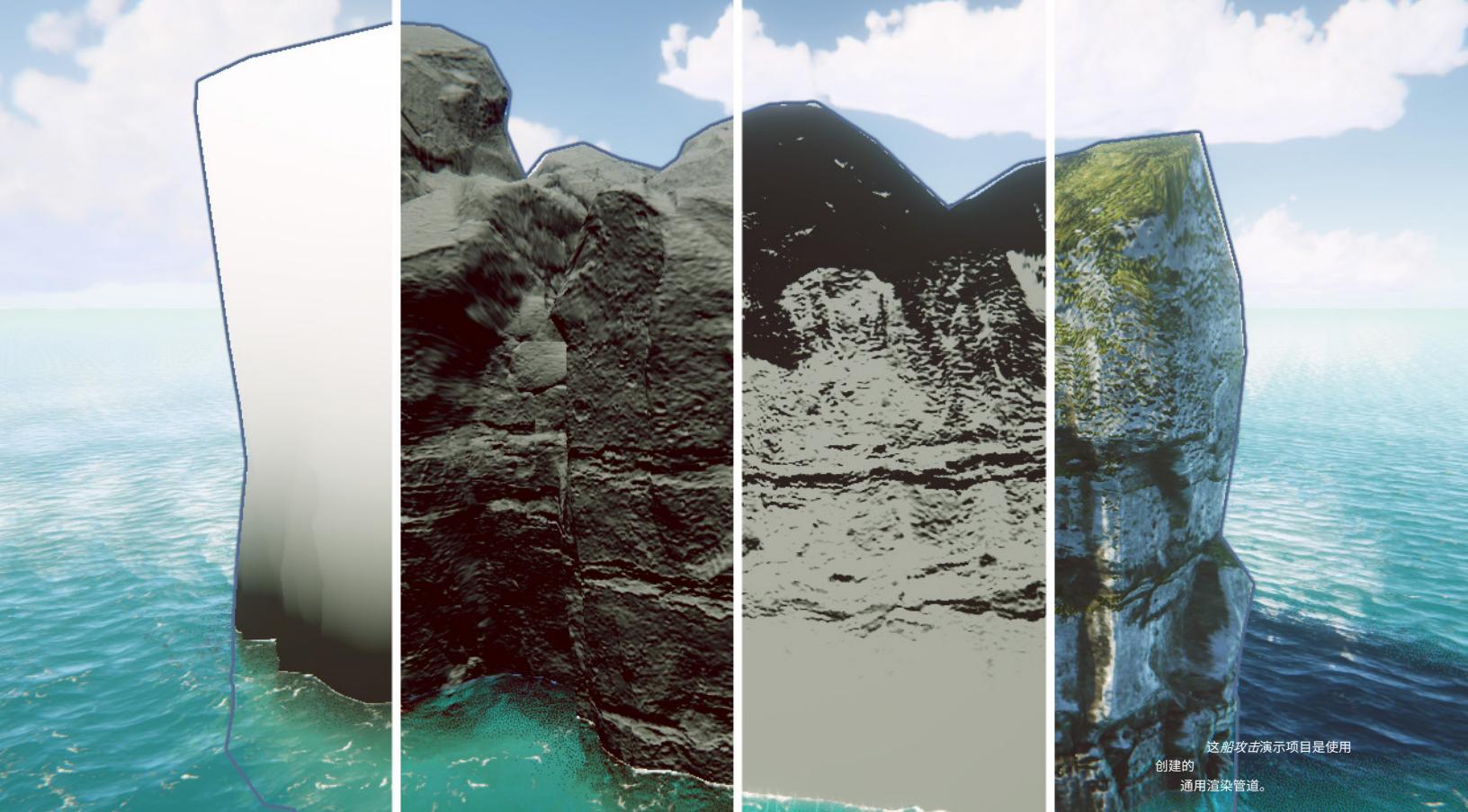
资产管道可以极大地影响应用程序的性能。经验丰富的技术美工可以帮助您的团队定义和实施资产格式、规范和导入设置。

不要依赖默认设置。使用特定于平台的覆盖选项卡来优化纹理和网格几何体等资源。不正确的设置可能会产生更大的构建大小、更长的构建时间和较差的内存使用率。考虑使用[预设](#)功能可帮助自定义特定项目的基线设置，以确保最佳设置。

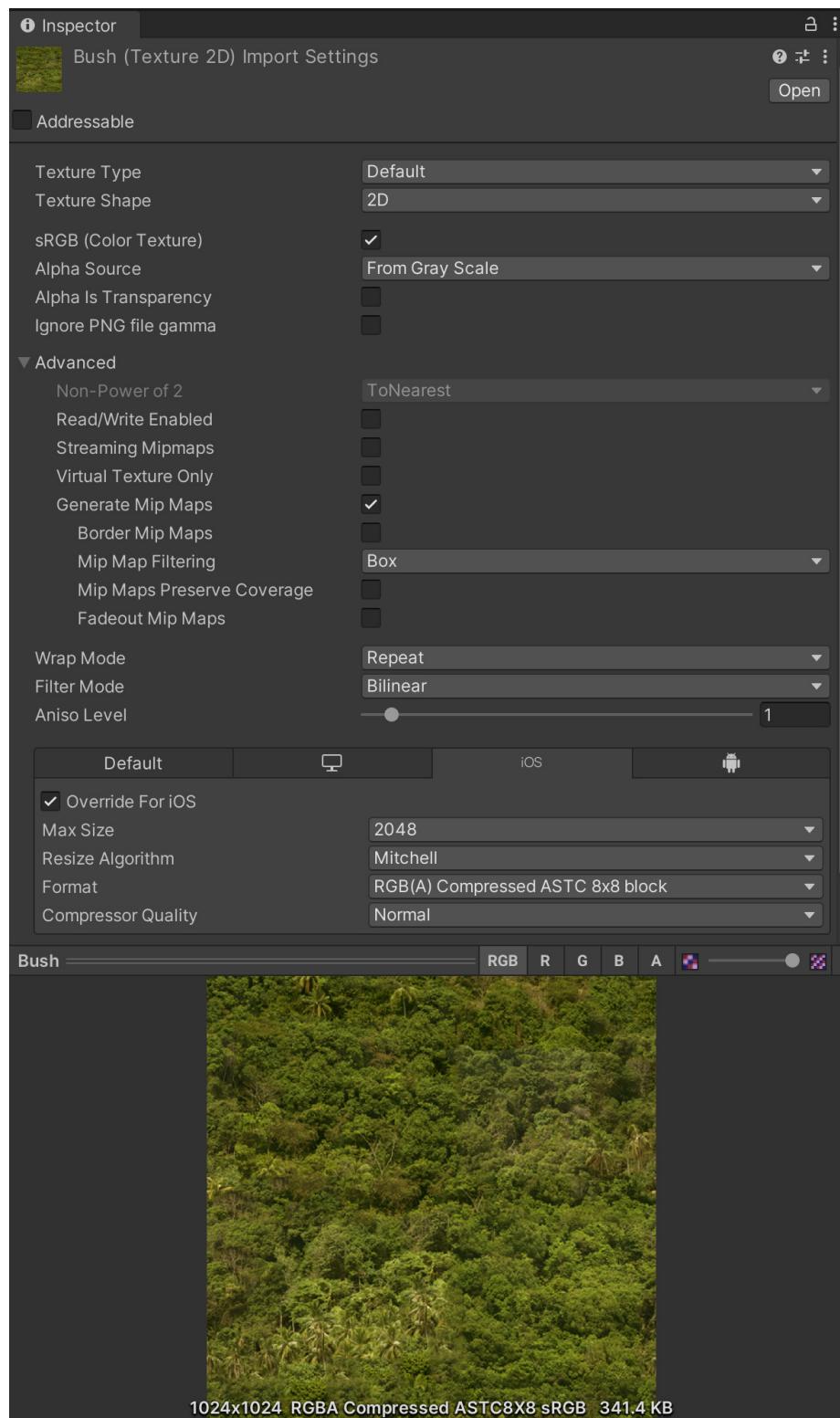
看[本艺术资产最佳实践指南](#)了解更多详情或查看本课程移动应用程序的3D艺术优化有关更多详细信息，请访问Unity Learn。

正确导入纹理

您的大部分内存可能会存储在纹理中，因此此处的导入设置至关重要。一般来说，请遵循以下准则：



- 降低最大尺寸：使用产生视觉上可接受的结果的最小设置。这是非破坏性的，可以快速减少你的纹理内存。
- 使用 2 的幂 (POT)：Unity 需要移动纹理压缩格式 (PVRCT 或 ETC) 的 POT 纹理尺寸。
- 绘制纹理图集：将多个纹理放入单个纹理中可以减少绘制调用并加快渲染速度。使用[Unity Sprite Atlas](#) 或第三方[纹理打包器](#)到图集你的纹理。
- 关闭“读/写启用”选项：启用后，此选项会在 CPU 和 GPU 可寻址内存中创建副本，从而使纹理的内存占用量加倍。在大多数情况下，请保持禁用状态。如果您在运行时生成纹理，请通过 Texture2D .Apply 强制执行此操作，并将 makeNoLongerReadable 设置为 true。
- 禁用不必要的 Mip 贴图：对于在屏幕上保持一致大小的纹理，例如 2D 精灵和 UI 图形，不需要 Mip 贴图（对于与相机距离不同的 3D 模型，启用 Mip 贴图）。



正确的纹理导入设置将有助于优化您的构建尺寸。

压缩纹理

考虑使用相同模型和纹理的这两个示例。左边的设置消耗的内存几乎是右边设置的八倍，但对视觉质量没有太大好处。



未压缩的纹理需要更多内存。

对 iOS 和 Android 使用自适应可扩展纹理压缩 (ATSC)。绝大多数正在开发的游戏都针对支持 ATSC 压缩的最低规格设备。

唯一的例外是：

- 针对 A7 或更低版本设备的 iOS 游戏（例如 iPhone 5、5S 等）
 - 使用 PVRTC
- 针对 2016 年之前的设备的 Android 游戏 – 使用 ETC2（爱立信纹理压缩）

如果 PVRTC 和 ETC 等压缩格式的质量不够高，并且目标平台不完全支持 ASTC，请尝试使用 16 位纹理而不是 32 位纹理。

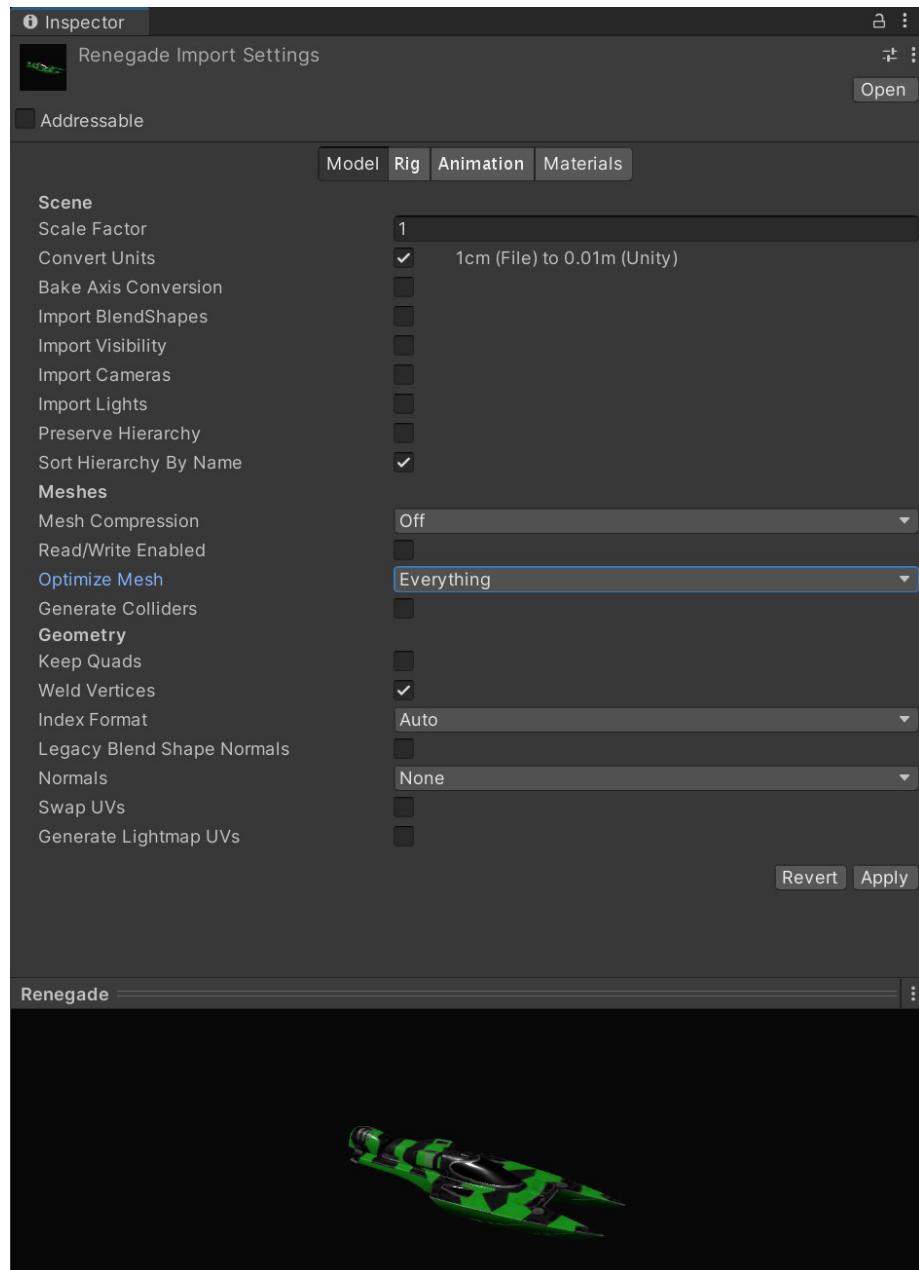
请参阅手册了解更多信息[按平台推荐的纹理压缩格式](#)。

调整网格导入设置

与纹理非常相似，如果不小心导入，网格会消耗过多的内存。要最大限度地减少网格的内存消耗：

- 压缩网格：积极的压缩可以减少磁盘空间（但是，运行时的内存不受影响）。请注意，网格量化可能会导致不准确，因此请尝试压缩级别，看看什么适合您的模型。

- 禁用读/写：启用此选项会在内存中复制网格，将网格的一个副本保留在系统内存中，将另一个副本保留在 GPU 内存中。在大多数情况下，您应该禁用它（在 Unity 2019.2 及更早版本中，默认情况下选中此选项）。
- 禁用装备和混合形状：如果您的网格不需要骨架或混合形状动画，请尽可能禁用这些选项。
- 如果可能，禁用法线和切线：如果您确定网格的材质不需要法线或切线，请取消选中这些选项以节省额外费用。



检查您的网格导入设置。

检查您的多边形数量

更高分辨率的模型意味着更多的内存使用和可能更长的 GPU 时间。您的背景几何形状需要五十万个多边形吗？考虑减少您选择的 DCC 封装中的型号。删除从相机的角度看不到的多边形。使用纹理和法线贴图来获得精细细节，而不是高密度网格。

使用 AssetPostprocessor 自动执行导入设置

这[资产后处理器](#)允许您在导入资产之前或导入资产时连接到导入管道并运行脚本。这会提示您在导入模型、纹理、音频等之前和/或之后以类似于预设的方式但通过代码自定义设置。在我们的 GDC 2023 演讲中了解有关该流程的更多信息，“[游戏创作每个阶段的技术提示](#)”。

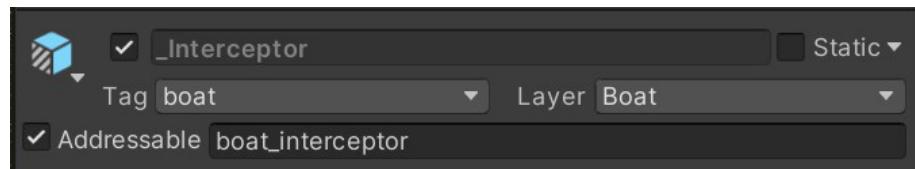
Unity数据工具

[Unity数据工具](#)是Unity提供的开源工具集合，旨在增强Unity项目中的数据管理和序列化能力。它包括用于分析和优化项目数据的功能，例如识别未使用的资产、检测资产依赖性以及减小构建大小。

使用可寻址资产系统

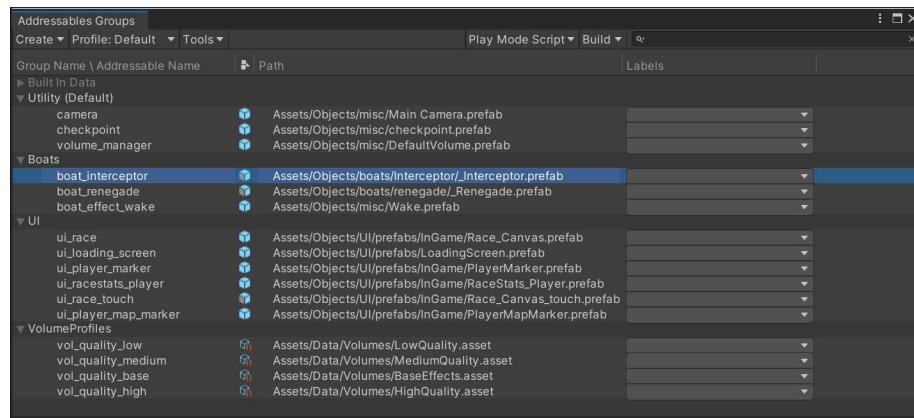
这[可寻址资产系统](#)提供了一种管理内容的简化方法，通过“地址”或别名加载 AssetBundles。该统一系统从本地路径或远程内容分发网络 (CDN) 异步加载。

如果您将非代码资源（模型、纹理、预制件、音频甚至整个场景）拆分为[资源包](#)，您可以将它们分开作为可下载的



内容 (DLC)。

然后，使用 Addressables 为您的移动应用程序创建较小的初始版本。云内容交付让您可以在玩家玩游戏时托管并交付游戏内容。



使用可寻址资产系统按“地址”加载资产。

点击[这里](#)了解可寻址资产系统如何减轻资产管理的痛苦。

图形和GPU优化

对于每一帧，Unity 都会确定必须渲染的对象，然后创建绘制调用。绘制调用是对图形 API 的调用以绘制对象（例如三角形），而批处理是一组要一起执行的绘制调用。

随着您的项目变得更加复杂，您将需要一个能够优化 GPU 上工作负载的管道。[通用渲染管线](#)(URP) 支持三种渲染选项：Forward、Forward+ 和 Deferred。



船攻击使用通用渲染管道创建的演示项目。

前向渲染会在一次传递中评估所有光照，通常建议将其作为手机游戏的默认设置。Unity 2022 LTS 中引入的 Forward+ 通过在空间上而不是按对象剔除灯光来改进标准前向渲染。这显着增加了可用于渲染帧的灯光总数。对于特定情况，例如具有大量动态光源的游戏，延迟模式是一个不错的选择。来自游戏机和 PC 的相同基于物理的照明和材质也可以扩展到您的手机或平板电脑。

下表比较了 URP 中的三个渲染选项。

特征	向前	转发+	延期
最大限度 数量 实时灯光 每个对象	9	无限; 的每 相机限制 适用	无限
每像素法线 编码	无编码 (准确的 正常值)	无编码 (准确的 正常值)	两种选择： <ul style="list-style-type: none">— G 缓冲区中法线的量化 (精度 损失, 性能更好)— 八面体编码 (准确的法线, 可能会对移动 GPU 产生显着的性能影响) 有关更多信息, 请参阅 G- buffer 中法线的编码 。
MSAA	是的	是的	不
顶点照明	是的	不	不
相机 堆叠	是的	是的	支持但有限制: Unity 仅使用 Deferred 路径渲染基础相机; Unity 使用前向渲染路径渲染所有 覆盖相机

通过电子书了解如何将基于内置渲染管道的项目迁移到 URP[面向高级 Unity 创作者的通用渲染
管道简介](#)。

GPU优化

要优化图形渲染，您需要了解目标硬件的限制以及如何分析 GPU。分析可帮助您检查并验证您所做的优化是否有效。

使用这些最佳实践来减少 GPU 上的渲染工作负载。

对 GPU 进行基准测试

进行分析时，从基准开始很有用。基准测试告诉您特定 GPU 应该得到什么样的分析结果。

看[GFX基准测试](#)查看 GPU 和显卡的不同行业标准基准的详细列表。该网站很好地概述了当前可用的 GPU 以及它们如何相互比较。

观看渲染统计数据

单击游戏视图右上角的统计按钮。此窗口向您显示有关应用程序在“播放”模式期间的实时渲染信息。使用此数据来帮助优化性能：

- fps：每秒帧数
- CPU Main：处理一帧的总时间（并更新所有窗口的编辑器）
- CPU 渲染：渲染游戏视图一帧的总时间
- 批次：一起绘制的绘制调用组
- Tris（三角形）和 Verts（顶点）：网格几何体
- SetPass 调用：Unity 必须切换着色器通道以在屏幕上渲染游戏对象的次数；每个通道都会引入额外的 CPU 开销。

注意：编辑器内 fps 并不一定转化为构建性能。我们建议您分析您的构建以获得最准确的结果。以毫秒为单位的帧时间是[比每秒帧数更准确的指标](#) 进行基准测试时。

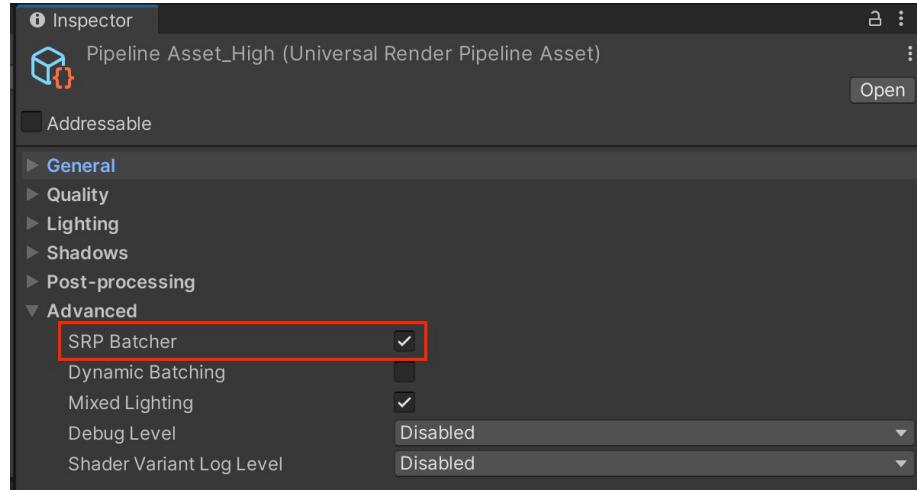
使用绘制调用批处理

为了绘制游戏对象，Unity 向图形 API（例如，OpenGL、Vulkan 或 Direct3D）发出绘制调用。每个绘制调用都是资源密集型的。绘制调用之间的状态变化（例如切换材质）可能会导致 CPU 端的性能开销。

PC 和控制台硬件可以推送大量绘制调用，但每次调用的开销仍然足够高，需要尝试减少它们。在移动设备上，绘制调用优化至关重要。你可以通过以下方式实现这一点[绘制调用批处理](#)。

绘制调用批处理可最大程度地减少这些状态更改并降低渲染对象的 CPU 成本。Unity 可以使用多种技术将多个对象合并为更少的批次：

- SRP 批处理：如果您使用 HDRP 或 URP，请启用[SRP批处理程序](#)在您的 Pipeline Asset 下的 Advanced 下。当使用兼容的着色器时，SRP Batcher 会减少绘制调用之间的 GPU 设置，并使材质数据持久保存在 GPU 内存中。这可以显着加快 CPU 渲染时间。少用点[着色器变体](#)用最少量的关键词来提高SRP批处理。咨询这个[SRP 文档](#)了解您的项目如何利用此渲染工作流程。



SRP批处理程序可以帮助您[批量绘制调用](#)。

- GPU 实例化：如果您有大量相同的对象（例如，具有相同网格和材质的建筑物、树木、草等），请使用[GPU实例化](#)。该技术使用图形硬件对它们进行批处理。要启用 GPU 实例化，请在“项目”窗口中选择材质，然后在检查器中选中“启用实例化”。

- 静态批处理：对于非移动几何体，Unity 可以减少共享相同材质的任何网格体的绘制调用。它比动态批处理更高效，但使用更多内存。

在检查器中将所有从不移动的网格标记为“批处理静态”。Unity 在构建时将所有静态网格物体组合成一个大网格物体。这[静态批处理实用程序](#)还允许您在运行时自己创建这些静态批次（例如，在生成非移动部件的程序级别之后）。

- 动态批处理：对于小网格，Unity可以在CPU上对顶点进行分组和变换，然后一次性将它们全部绘制出来。注意：做不是除非您有足够的低多边形网格（每个顶点不超过 300 个，总顶点属性不超过 900 个），否则请使用此选项。否则，启用它会浪费 CPU 时间来寻找小网格进行批处理。

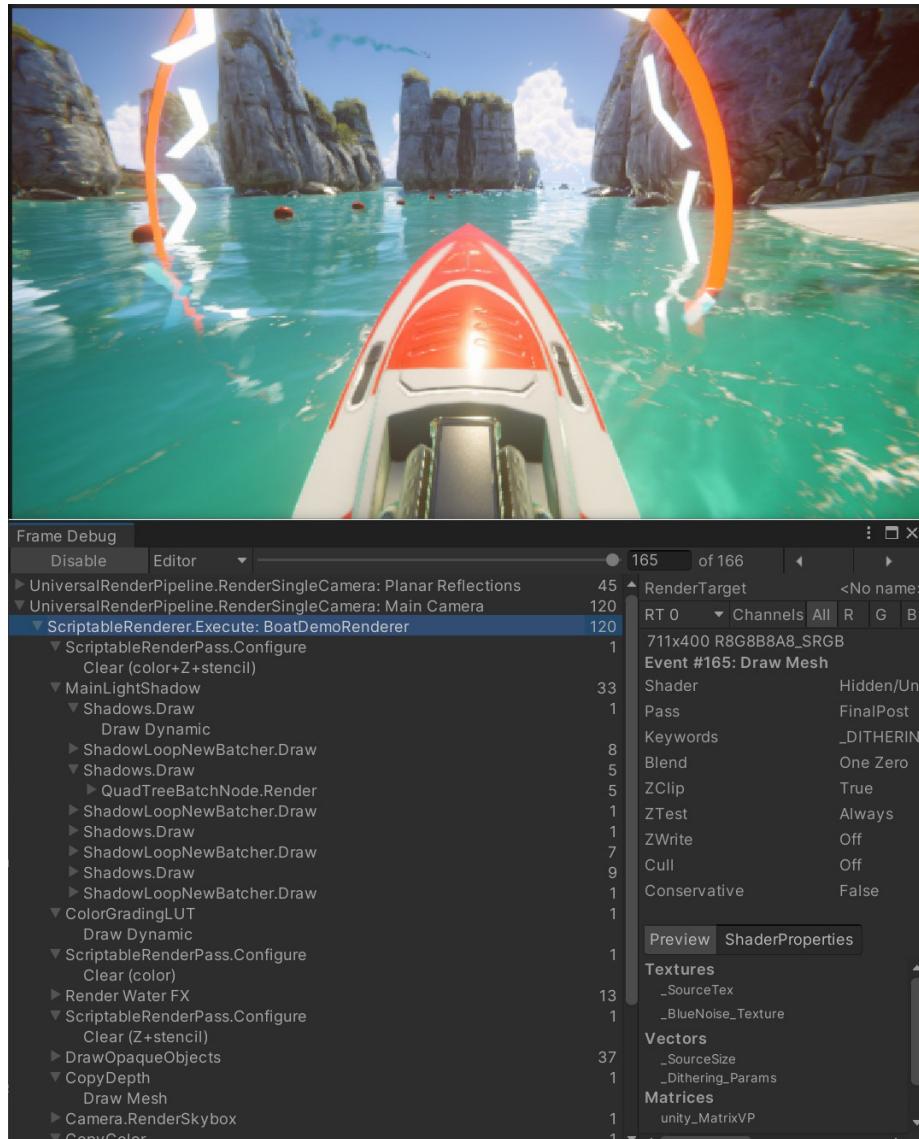
您可以通过一些简单的规则来最大化批处理：

- 在场景中使用尽可能少的纹理。更少的纹理需要更少的独特材料，使它们更容易批处理。此外，尽可能使用纹理图集。
- 始终以尽可能最大的图集尺寸烘焙光照贴图。较少的光照贴图需要较少的材质状态更改，但请注意内存占用。
- 请注意不要无意中实例化材料。
访问[渲染器.material](#)在脚本中复制材料并返回对新副本的引用。这会破坏任何已包含该材料的现有批次。如果您想访问批处理对象的材质，请使用[渲染器.sharedMaterial](#)反而。
- 通过在优化过程中使用探查器或渲染统计信息，密切关注静态和动态批次计数与总绘制调用计数的关系。

请参阅[绘制调用批处理](#)文档以获取更多信息。

使用帧调试器

这**帧调试器**显示每个帧是如何通过单独的绘制调用构建的。这是解决着色器属性问题的宝贵工具，可以帮助您分析游戏的渲染方式。



帧调试器将每个帧分解为单独的步骤。

帧调试器新手？看看这篇介绍[这里](#)。

避免过多的动态灯光

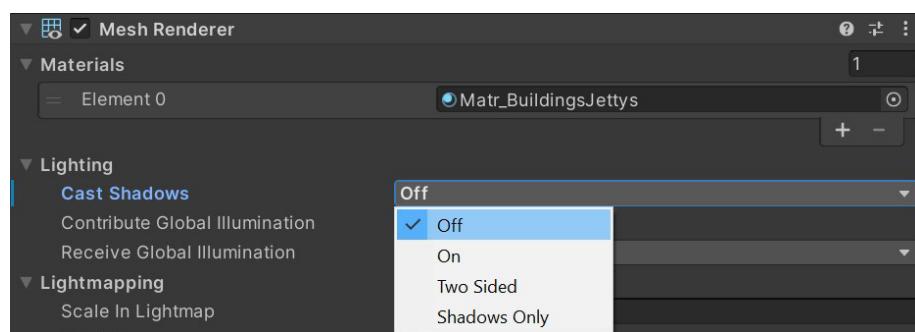
使用前向渲染时，避免向移动应用程序添加太多动态光源至关重要。考虑替代方案，例如自定义着色器效果和动态网格物体的光探针，以及静态网格物体的烘焙照明。

看到这个[功能比较表](#)针对 URP 和内置渲染管线实时灯光的具体限制。

禁用阴影

可以根据 MeshRenderer 和 light 禁用阴影投射。尽可能禁用阴影以减少绘制调用。

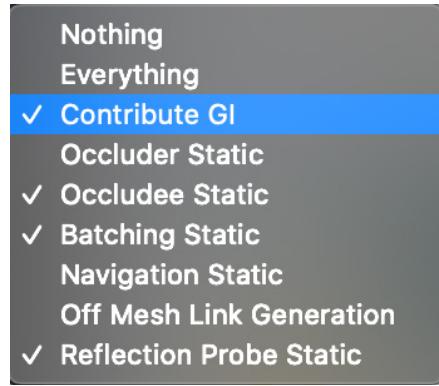
您还可以使用应用于角色下方的简单网格或四边形的模糊纹理来创建假阴影。或者，使用自定义着色器创建斑点阴影。



禁用阴影投射以减少绘制调用。

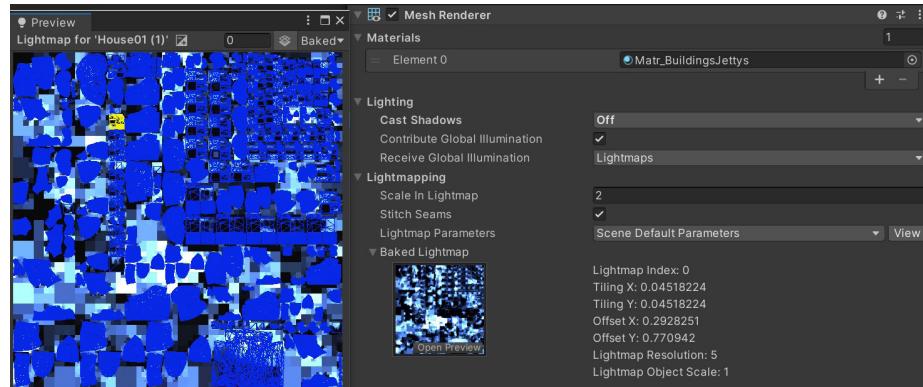
将您的光照烘焙到光照贴图中

使用全局照明 (GI) 为静态几何体添加戏剧性的照明。使用 Contribute GI 标记对象，以便您可以以 Lightmaps 的形式存储高质量的光照。



启用贡献 GI。

然后可以渲染烘焙的阴影和光照，而不会影响运行时的性能。Progressive CPU 和 GPU Lightmapper 可以加速全局照明的烘焙。

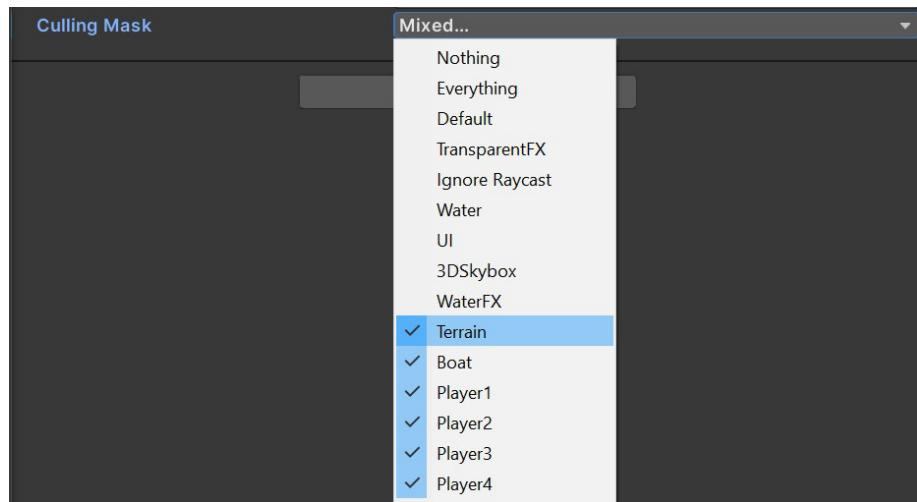


调整光照贴图设置 (Windows > 渲染 > 光照设置) 和光照贴图大小以限制内存使用。

跟着[手动引导](#)和[这篇关于光优化的文章](#)开始使用 Unity 中的光照贴图。

使用光层

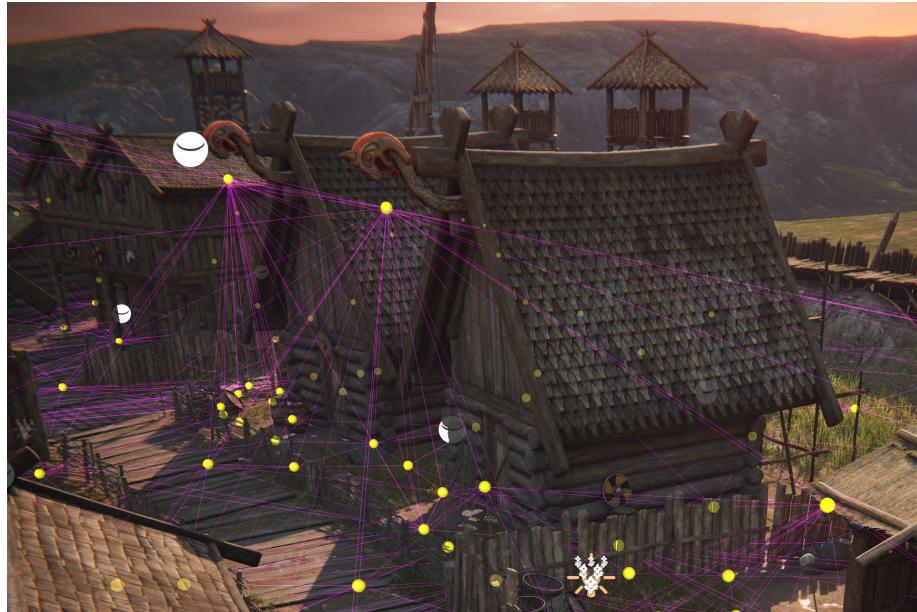
对于具有多个灯光的复杂场景，请使用图层分隔对象，然后将每个灯光的影响限制在特定的剔除蒙版中。



图层可以限制光线对特定剔除蒙版的影响。

使用光探针检测移动物体

光探针存储有关场景中空白空间的烘焙光照信息，同时提供高质量的光照（直接和间接）。他们使用球谐函数，与动态灯相比，计算速度更快。这对于通常无法接收烘焙光照贴图的移动对象特别有用。



一个带有光探针的光探针组遍布整个关卡。

光探针也可以应用于静态网格物体。在 MeshRenderer 组件中，找到 Receive Global Illumination 下拉列表，并将其从 Lightmaps 切换到 Light Probes。

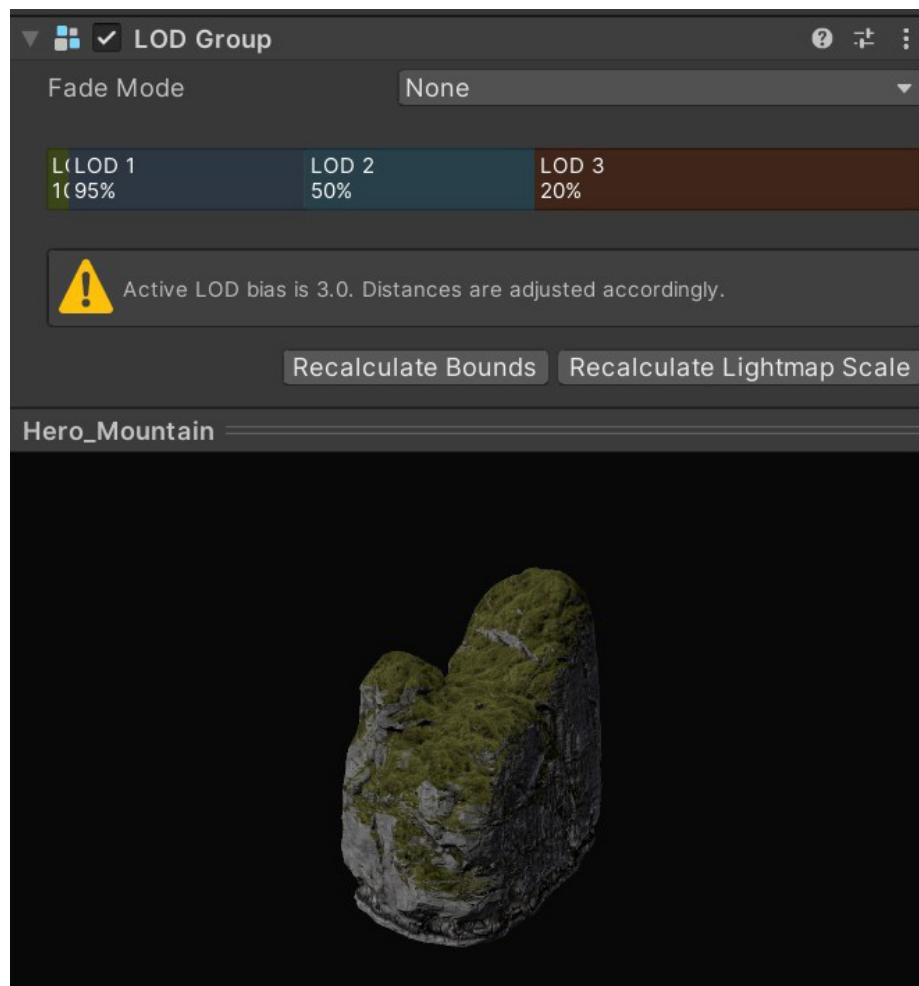
继续对突出的关卡几何体使用光照贴图，但对较小的细节使用探针。光探针照明不需要适当的 UV，从而节省了展开网格的额外步骤。探针还可以减少磁盘空间，因为它们不生成光照贴图纹理。

参见“[使用光探针的静态照明](#)”博客文章，了解有关使用灯光选择性照亮场景对象的信息探头。

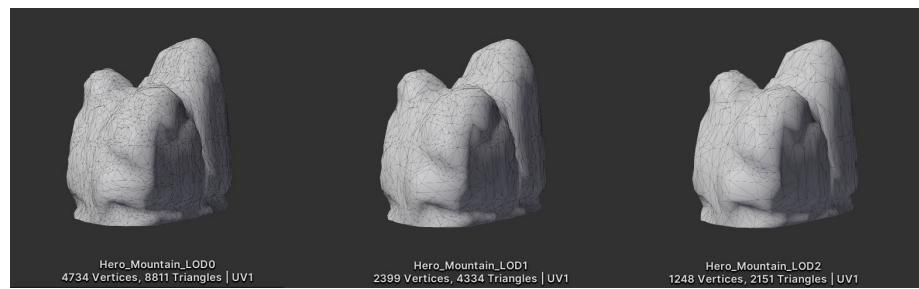
有关 Unity 中的光照工作流程的更多信息，请阅读“[在 Unity 中制作可信的视觉效果。](#)”

使用细节级别 (LOD)

当物体移向远处时，**详细程度**(LOD) 可以将它们切换为使用更简单的网格以及更简单的材质和着色器，以提高 GPU 性能。



使用 LOD 组的网格示例。



源网格，以不同的分辨率建模。

请参阅[使用 LOD](#)有关更多详细信息，请参阅 Unity Learn 课程。

使用遮挡剔除删除隐藏的对象

隐藏在其他对象后面的对象仍然可能会渲染并消耗资源。使用遮挡剔除来丢弃它们。

虽然相机视图之外的视锥体剔除是自动的，但遮挡剔除是一个烘焙过程。只需将对象标记为静态遮挡物或被遮挡物，然后通过 Window > Rendering > Occlusion Culling 进行烘焙。尽管并非每个场景都需要剔除，但剔除可以在特定情况下提高性能，因此请务必在启用遮挡剔除之前和之后进行分析，以检查其是否提高了性能。

查看[使用遮挡剔除](#)教程以获取更多信息。

避免移动原生分辨率

手机和平板电脑变得越来越先进，更新的设备具有非常高的分辨率。

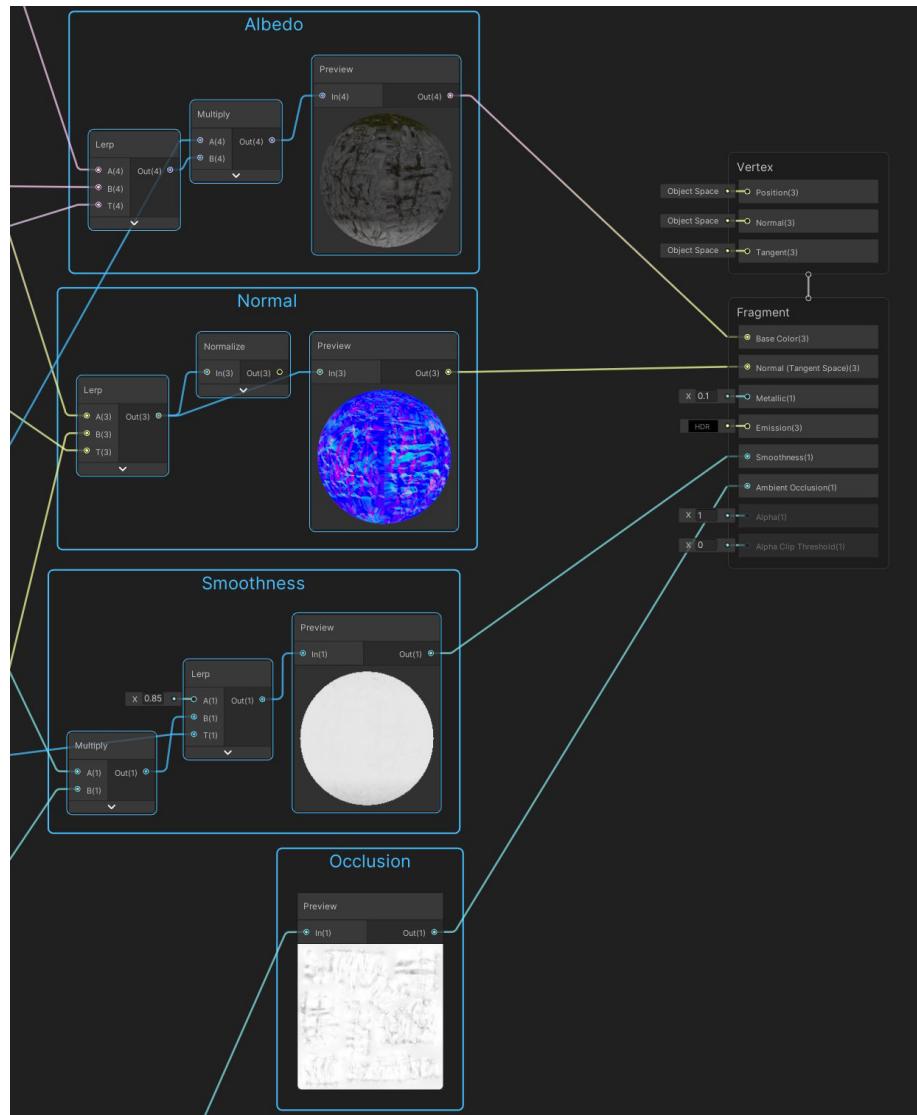
使用 Screen .SetResolution(width, height, false) 降低输出分辨率并恢复一些性能。配置多种分辨率以找到质量和速度之间的最佳平衡。

限制相机的使用

每个启用的相机都会产生一些开销，无论它是否在做有意义的工作。仅使用渲染所需的相机组件。在低端移动平台上，每个摄像头最多可以使用 1 毫秒的 CPU 时间。

保持着色器简单

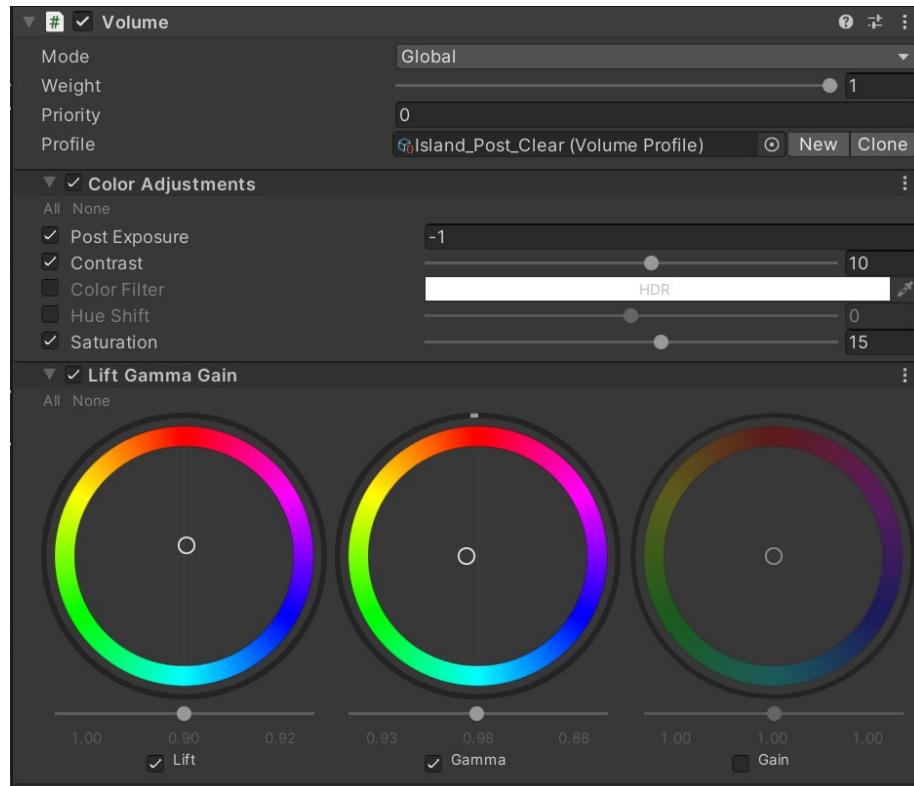
通用渲染管道包括多个轻量级 Lit 和 Unlit 着色器，这些着色器已经针对移动平台进行了优化。尝试使着色器变化尽可能低，因为它们会对运行时内存使用产生巨大影响。如果默认的 URP 着色器不能满足您的需求，您可以使用 Shader Graph 自定义材质的外观。了解如何使用 Shader Graph 直观地构建着色器[这里](#)。



使用 Shader Graph 创建自定义着色器。

最大限度地减少过度绘制和 Alpha 混合

避免绘制不必要的透明或半透明图像，并且不要重叠几乎不可见的图像或效果。移动平台受到由此产生的过度绘制和 alpha 混合的极大影响。您可以使用以下命令检查透支[渲染文档](#)图形调试器。您还可以利用[渲染调试器](#)，它可以让您可视化各种照明、渲染和材质属性。可视化可帮助您识别渲染问题并优化场景和渲染配置。



在移动应用程序中保持后期处理效果简单。

限制后处理效果

全屏[后期处理](#)像发光这样的效果会大大降低性能。在标题的艺术指导下谨慎使用它们。

小心渲染器 .material

在脚本中访问 `Renderer.material` 会复制材质并返回对新副本的引用。这会破坏任何已包含该材料的现有批次。如果您想访问批处理对象的材质，请使用[渲染器.sharedMaterial](#)反而。

优化蒙皮网格渲染器

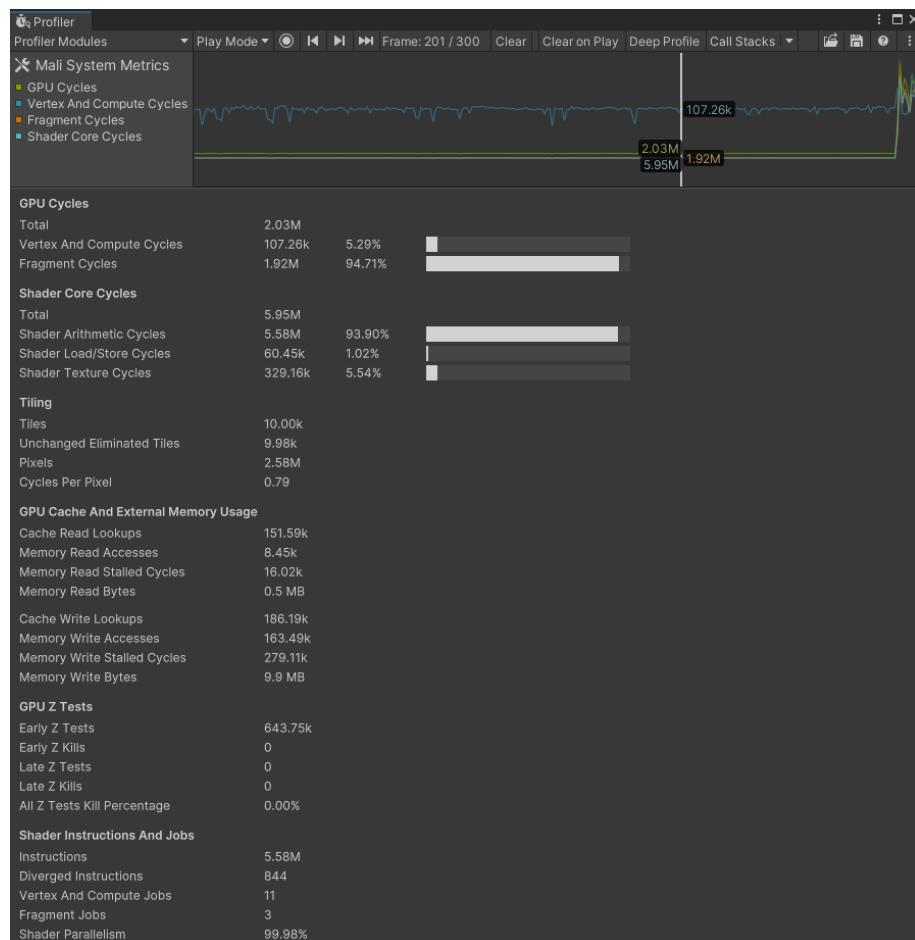
渲染蒙皮网格的成本很高。确保使用 SkinnedMeshRenderer 的每个对象都需要它。如果游戏对象仅在某些时候需要动画，请使用 BakeMesh 函数将蒙皮网格冻结在静态姿势，并在运行时切换到更简单的 MeshRenderer。

尽量减少反射探头

A [反射探头](#)可以创建逼真的反射，但这在批量方面可能会非常昂贵。使用低分辨率立方体贴图、剔除蒙版和纹理压缩来提高运行时性能。

系统指标 马里

您还可以利用 System Metrics Mali 软件包来访问使用 ARM GPU 的设备上的低级系统或硬件性能指标。这包括能够在 Unity Profiler 中监控低级 GPU 指标、使用 Recorder API 在运行时访问低级 GPU 指标，以及通过持续集成测试运行自动执行性能测试。



Mali 系统指标分析器模块

用户界面

Unity 提供两种 UI 系统，旧版 Unity UI 和新版 Unity UI[用户界面工具包](#)。UI Toolkit 旨在成为推荐的 UI 系统。它专为实现最高性能和可重用性而量身定制，其工作流程和创作工具受到标准 Web 技术的启发，这意味着 UI 设计师和艺术家如果已经有设计网页的经验，就会发现它很熟悉。

但是，从 Unity 2022 LTS 开始，UI Toolkit 不具备某些功能[统一用户界面](#)和[立即模式 GUI \(IMGUI\)](#)支持。Unity UI 和 IMGUI 更适合某些用例，并且需要支持遗留项目。请参阅[Unity 中 UI 系统比较](#)了解更多信息。

UGUI性能优化技巧

[统一用户界面\(UGUI\)](#) 通常是性能问题的根源。Canvas 组件生成并更新 UI 元素的网格并向 GPU 发出绘制调用。它的运行成本可能很高，因此在使用 UGUI 时请记住以下因素。

划分你的画布

如果您有一个包含数千个元素的大型 Canvas，则更新单个 UI 元素会强制整个 Canvas 更新，从而可能会产生 CPU 峰值。

利用UGUI支持多个Canvas的能力。根据 UI 元素需要刷新的频率来划分它们。将静态 UI 元素保留在单独的 Canvas 上，并将同时更新的动态元素保留在较小的子画布上。

确保每个 Canvas 中的所有 UI 元素都具有相同的 Z 值、材质和纹理。

隐藏不可见的 UI 元素

您可能拥有仅在游戏中偶尔出现的 UI 元素（例如，仅在角色受到伤害时出现的生命条）。

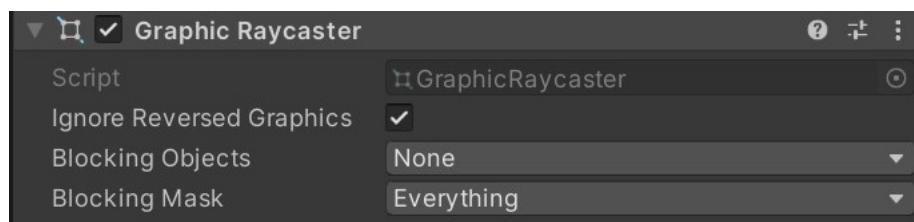
如果您的不可见 UI 元素处于活动状态，则它可能仍在使用绘制调用。显式禁用任何不可见的 UI 组件并根据需要重新启用它们。

如果您只需要关闭 Canvas 的可见性，请禁用 Canvas 组件而不是整个 GameObject。这可以防止您的游戏在重新启用它时必须重建网格和顶点。

限制 GraphicRaycasters 并禁用 Raycast Target

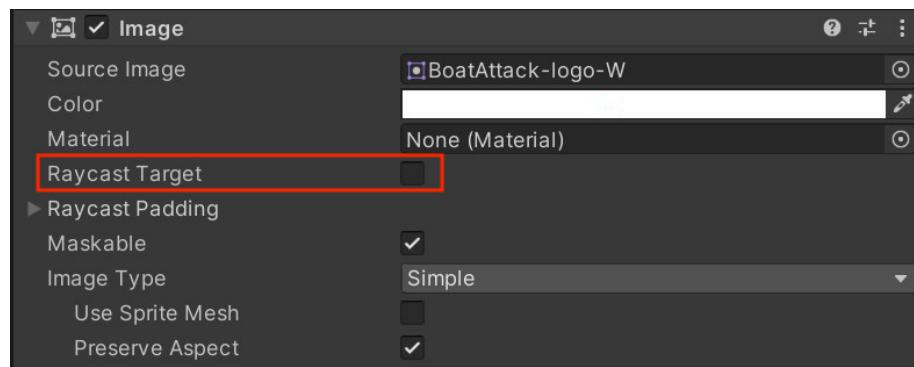
屏幕触摸或点击等输入事件需要 GraphicRaycaster 组件。这只是循环遍历屏幕上的每个输入点并检查它是否在 UI 的 RectTransform 内。

从层次结构中的顶部 Canvas 中删除默认的 GraphicRaycaster。相反，将 GraphicRaycaster 专门添加到需要交互的各个元素（按钮、滚动矩形等）。



禁用“忽略反转图形”，该选项默认处于活动状态。

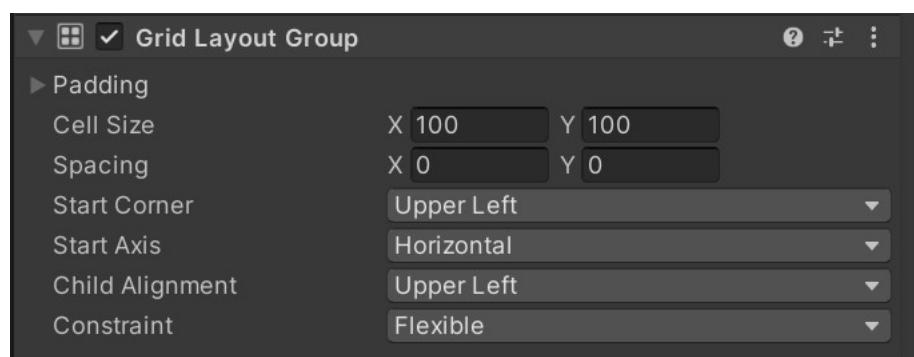
此外，在所有不需要的 UI 文本和图像上禁用 Raycast Target。如果 UI 很复杂，有很多元素，所有这些小的改变都可以减少不必要的计算。



避免布局组

布局组更新效率低下，因此请谨慎使用。如果您的内容不是动态的，请完全避免它们，并使用锚点进行比例布局。或者，创建自定义代码以禁用[布局组](#)设置 UI 后的组件。

如果您确实需要对动态元素使用布局组（水平、垂直、网格），请避免嵌套它们以提高性能。



布局组会降低性能，尤其是在嵌套时。

避免使用大型列表和网格视图

大型列表和网格视图的成本很高。如果您需要创建大型列表或网格视图（例如，包含数百个项目的库存屏幕），请考虑重用较小的 UI 元素池，而不是为每个项目创建一个 UI 元素。查看此示例[GitHub 项目](#)来看看它的实际效果。

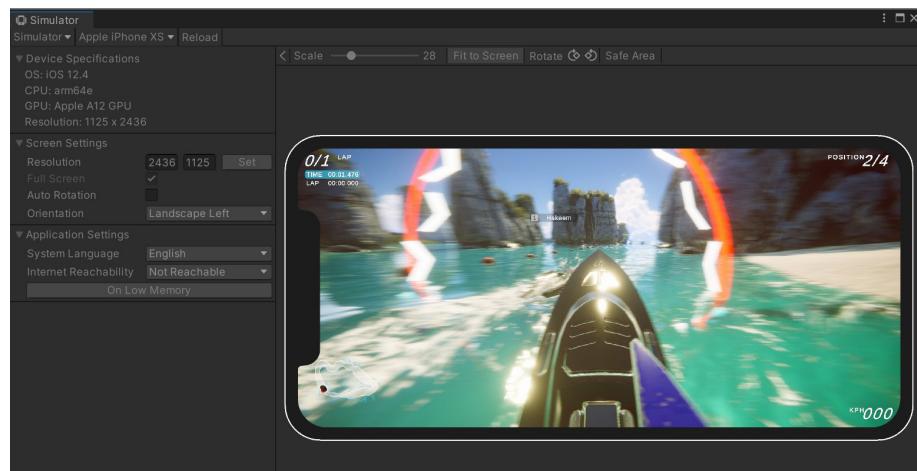
避免大量重叠的元素

将大量 UI 元素分层（例如，纸牌战斗游戏中堆叠的纸牌）会造成透支。自定义您的代码以在运行时将分层元素合并为更少的元素和批次。

使用多种分辨率和宽高比

随着移动设备现在使用截然不同的分辨率和屏幕尺寸，创建[UI 的替代版本](#)为每台设备提供最佳体验。

使用设备模拟器预览各种受支持设备的 UI。您还可以在中创建虚拟设备[代码](#)和[安卓工作室](#)。



使用设备模拟器预览各种屏幕格式。

使用全屏 UI 时，隐藏其他所有内容

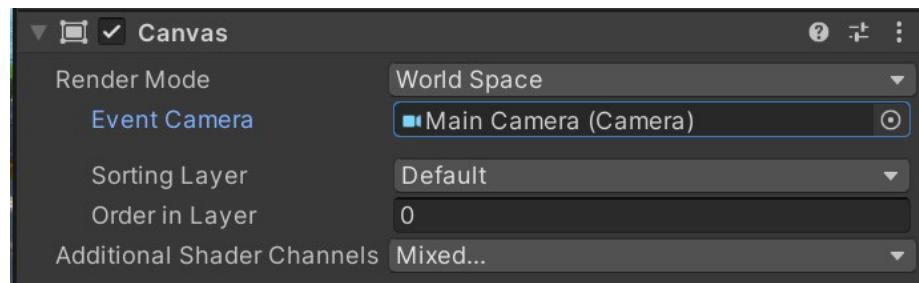
如果您的暂停屏幕或开始屏幕覆盖了场景中的其他所有内容，请禁用渲染 3D 场景的相机。同样，禁用隐藏在顶部 Canvas 后面的任何背景 Canvas 元素。

考虑在全屏 UI 期间降低 Application .targetFrameRate，因为您不需要以 60 fps 进行更新。

将相机分配给世界空间和相机空间画布

将 Event 或 Render Camera 字段留空会强制 Unity 填充 Camera .main，这会导致不必要的昂贵开销。

如果可能的话，请考虑对 Canvas RenderMode 使用“屏幕空间 - 覆盖”(Screen Space – Overlay)，因为这不需要相机。



使用世界空间渲染模式时，请确保填写事件相机。

UI Toolkit 性能优化技巧

UI Toolkit 提供比 Unity UI 更高的性能，专为最大性能和可重用性而定制，并提供受标准 Web 技术启发的工作流程和创作工具。其主要优点之一是它使用专为 UI 元素设计的高度优化的渲染管道。

以下是使用 UI Toolkit 优化 UI 性能的一些一般建议：

- 使用高效布局：高效布局是指使用[布局组](#)由UI Toolkit (例如Flexbox) 提供，而不是手动定位和调整UI元素的大小。布局组自动处理布局计算，这可以显着提高性能。它们确保 UI 元素根据指定的布局规则正确排列和调整大小。通过利用高效的布局，您可以避免手动布局计算的开销，并实现一致且优化的 UI 渲染。
- 避免 Update 中昂贵的操作：最大限度地减少 Update 方法中执行的工作量，尤其是 UI 元素创建、操作或计算等繁重操作。请谨慎执行这些操作，或者尽可能在初始化期间执行这些操作，因为更新方法每帧调用一次。
- 优化事件处理：留意事件订阅，并在不再需要时取消注册它们。过多的事件处理可能会影响性能，因此请确保您只订阅必要的事件。

- 优化样式表：注意样式表中使用的样式类和选择器的数量。具有大量规则的大型样式表会影响性能。保持样式表简洁并避免不必要的复杂性。
- 分析和优化：使用 Unity 的分析工具来识别 UI 中的性能瓶颈，并找出可以进一步优化的区域，例如低效的布局计算或过多的重绘。
- 在目标平台上测试：在目标平台上测试您的UI性能，以确保在不同设备上获得最佳性能。性能可能会因硬件功能而异，因此在优化 UI 时请考虑目标平台。

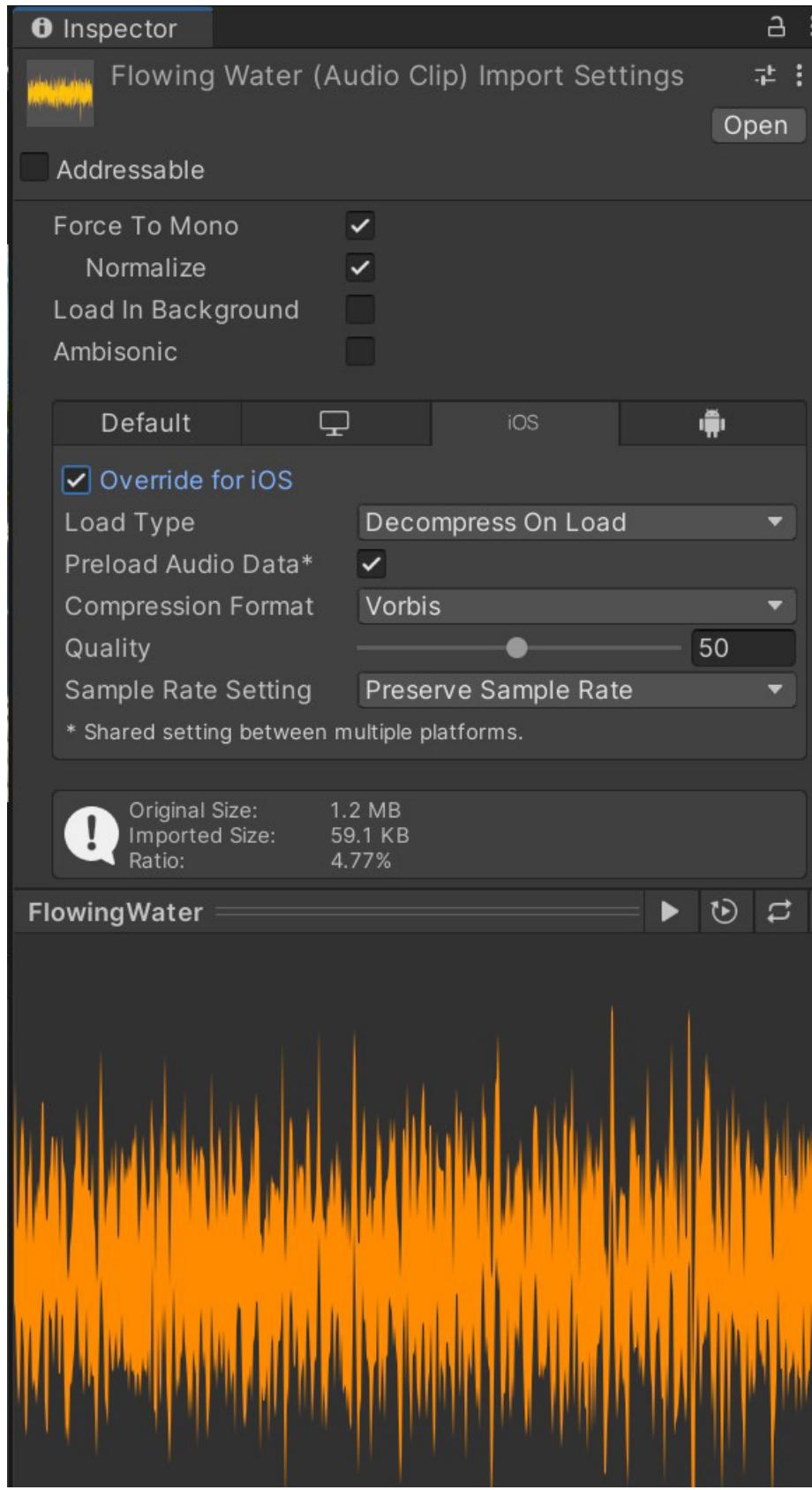
请记住，性能优化是一个迭代过程。持续分析、测量和优化您的 UI 代码，以确保其平稳高效地运行。

声音的

尽管音频通常不是性能瓶颈，但您仍然可以进行优化以节省内存。

尽可能使声音剪辑为单声道

如果您使用 3D 空间音频，请将声音剪辑创作作为单声道（单声道）或启用“强制为单声道”设置。在运行时定位使用的多声道声音将被扁平化为单声道源，从而增加 CPU 成本并浪费内存。



优化 AudioClip 的导入设置。

尽可能使用原始未压缩的 WAV 文件作为源资源

如果您使用任何压缩格式（例如 MP3 或 Vorbis），则 Unity 将对其进行解压缩并在构建时重新压缩。这会导致两次有损传递，从而降低最终质量。

压缩剪辑并降低压缩比特率

通过压缩减少剪辑的大小和内存使用量：

- 对于大多数声音，请使用 Vorbis（对于不打算循环的声音，请使用 MP3）。
- 使用 ADPCM 来播放简短的、常用的声音（例如脚步声、枪声）。与未压缩的 PCM 相比，这会缩小文件，但速度更快
在播放期间解码。

移动设备上的音效最高应为 22,050 Hz。使用较低的设置通常对最终质量的影响最小；你自己的耳朵可以帮助你自己判断。

选择合适的负载类型

该设置因剪辑尺寸而异。

- 小剪辑 (< 200 kb) 应在加载时解压缩。通过将声音解压缩为原始 16 位 PCM 音频数据，这会产生 CPU 成本和内存，因此仅适用于短声音。
- 中型剪辑 (>= 200 kb) 应在内存中保持压缩状态。
- 大文件（背景音乐）应设置为 Streaming。否则，整个资源将被一次性加载到内存中。

从内存中卸载静音的音频源

实现静音按钮时，不要简单地将音量设置为 0。您可以销毁 AudioSource 组件以将其从内存中卸载，前提是播放器不需要经常打开和关闭此组件。

动画片

统一的[麦克尼姆系统](#)是相当精密的。如果可能，请使用以下设置限制您在移动设备上的使用。

使用通用装备与人形装备

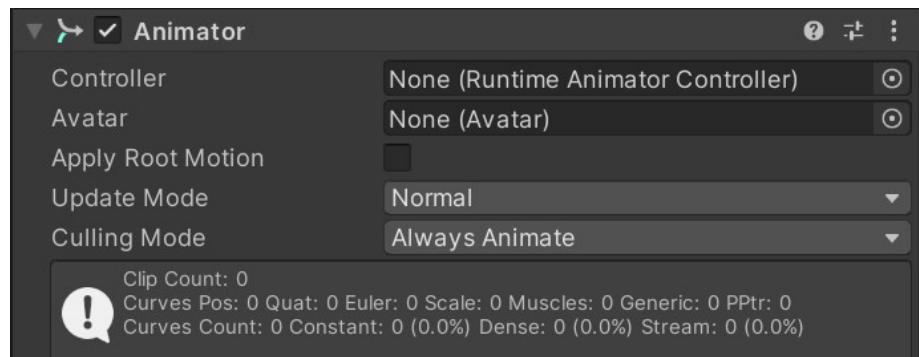
默认情况下，Unity 使用 Generic Rig 导入动画模型，但开发人员在为角色制作动画时经常切换到 Humanoid Rig。

人形装备比同等的通用装备多消耗 30-50% 的 CPU 时间，因为它计算反向运动学和动画重定向每帧，即使在不使用时也是如此。如果您不需要人形装备的这些特定功能，请改用通用装备。

避免过度使用动画师

动画器主要用于人形角色，但通常用于对单个值进行动画处理（例如，UI 元素的 Alpha 通道）。[避免过度使用动画师](#)，特别是与 UI 元素结合使用。只要有可能，请使用移动设备的旧版动画组件。

考虑创建补间函数或使用第三方库来实现简单动画（例如 DOTween）。



动画师的费用可能很高。

物理

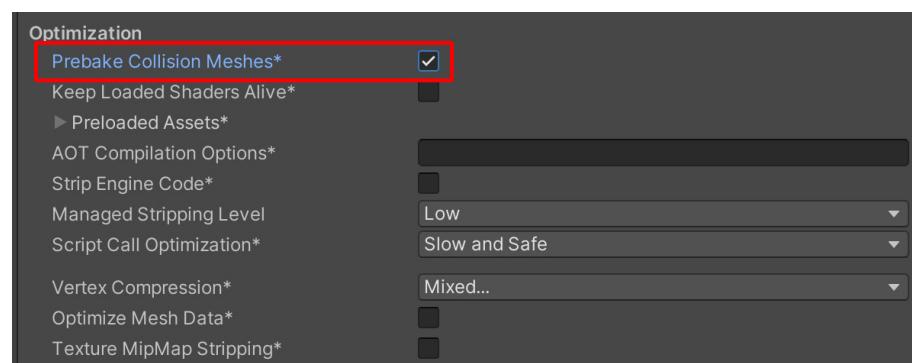
Unity 的内置物理 (Nvidia PhysX) 在移动设备上可能会很昂贵。以下提示可能会帮助您每秒挤出更多帧。

优化您的设置

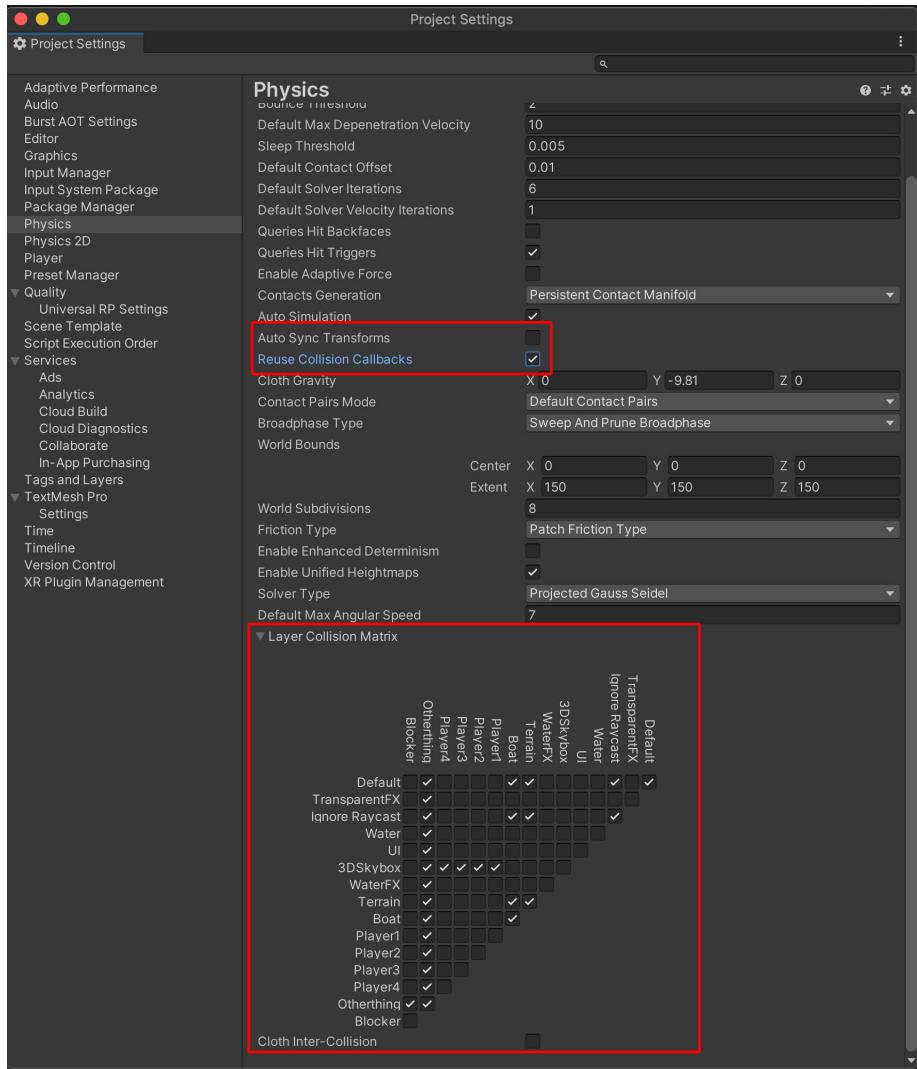
在里面[玩家设置](#)尽可能检查预烘焙碰撞网格。

确保您还编辑了物理设置（项目设置 > 物理）。尽可能简化层碰撞矩阵。

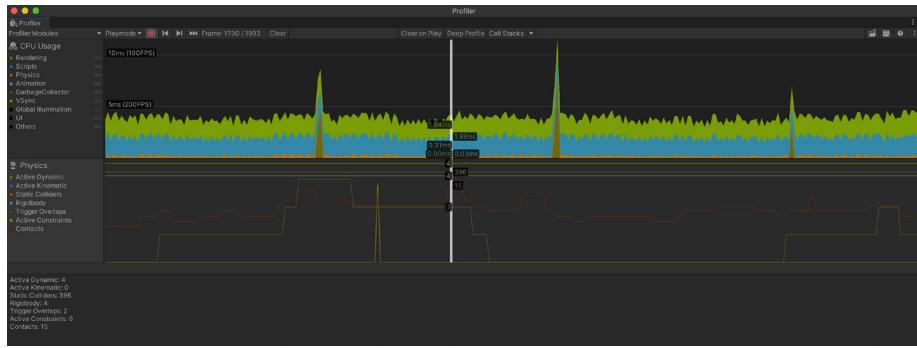
禁用自动同步变换并启用重用碰撞回调。



启用预烘焙碰撞网格。



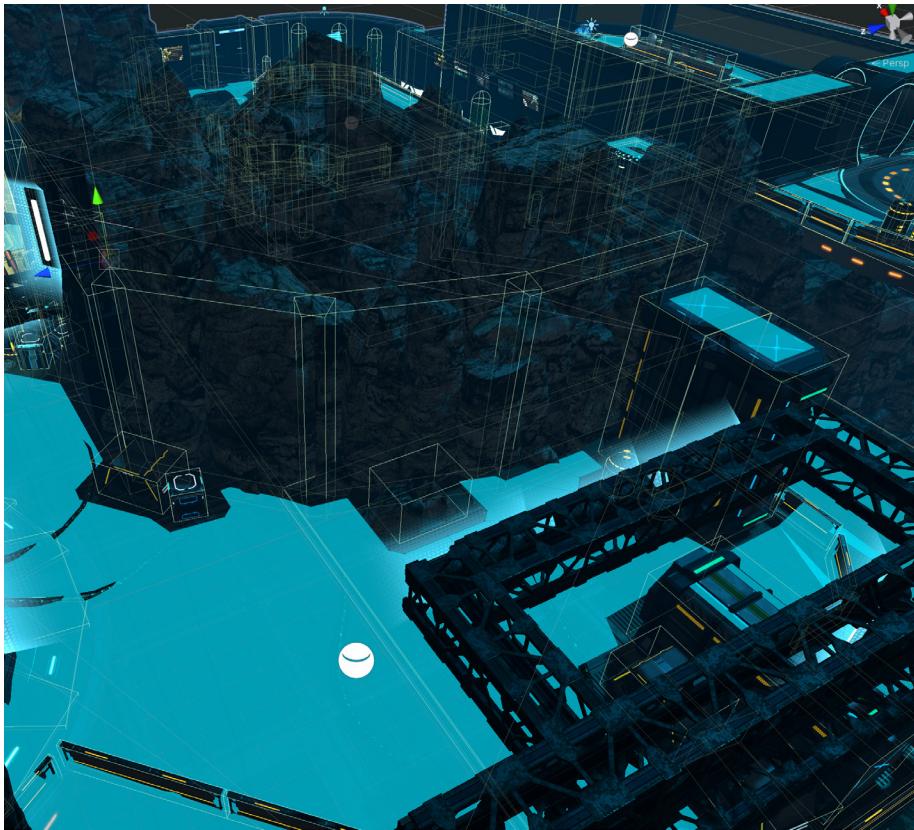
修改物理项目设置以挤出更多性能。



密切关注**物理模块**Profiler 的性能问题。

简化碰撞器

网格碰撞器可能很昂贵。用更简单的图元或网格碰撞器替换更复杂的网格碰撞器以近似原始形状。



使用基本体或简化的网格作为碰撞器。

使用物理方法移动刚体

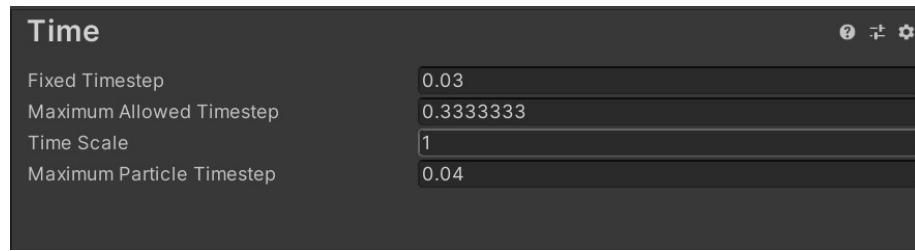
使用 `MovePosition` 或 `AddForce` 等类方法来移动 `Rigidbody` 对象。直接平移它们的 `Transform` 组件可能会导致物理世界重新计算，这在复杂场景中可能会很昂贵。在 `FixUpdate` 而不是 `Update` 中移动物理体。

修复固定时间步长

项目设置中的默认固定时间步长为 0.02 (50 Hz)。更改此值以匹配您的目标帧速率（例如 0.03 表示 30 fps）。

否则，如果您的帧速率在运行时下降，则意味着 Unity 会在每帧多次调用 `FixUpdate`，这可能会导致物理密集内容的 CPU 性能问题。

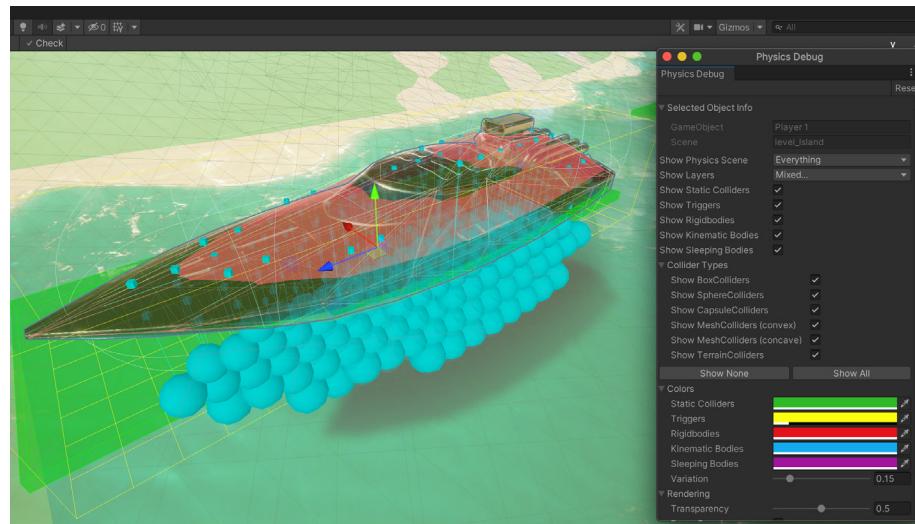
最大允许时间步长限制了在帧速率下降的情况下物理计算和固定更新事件可以使用的时间量。降低此值意味着物理和动画可以减慢速度，同时还可以减少性能故障期间对帧速率的影响。



修改固定时间步长以匹配您的目标帧速率，并降低最大允许时间步长以减少性能故障。

使用物理调试器进行可视化

使用“物理调试”窗口（“窗口”>“分析”>“物理调试器”）帮助解决任何碰撞器问题或差异。这显示了一个颜色编码指示器，指示哪些游戏对象应该能够相互碰撞。



物理调试器可帮助您直观地了解物理对象如何相互交互。

有关更多信息，请参阅[物理调试可视化](#)在Unity文档中。

工作流程和协作

在 Unity 中构建应用程序是一项艰巨的任务，通常会涉及许多开发人员。确保您的项目针对您的团队进行了最佳设置。

使用版本控制

每个人都应该使用某种类型的版本控制。确保您的编辑器设置将 Asset Serialization Mode 设置为 Force Text。



如果您在中使用外部版本控制系统（例如 Git）
版本控制设置，确保模式设置为可见元文件。



Unity还有一个内置的YAML（一种人类可读的数据序列化语言）工具，专门用于合并场景和预制件。有关更多信息，请参阅[智能合并](#)在 Unity 文档中。

版本控制对于团队工作至关重要。它可以帮助您追踪错误和错误的修订。遵循良好实践，例如使用分支和标签来管理里程碑和版本。

如需进一步的版本控制支持，请查看[塑料单片机](#)，我们推荐的 Unity 游戏开发版本控制解决方案。

分解大场景

大型、单一的 Unity 场景不太适合协作。将您的关卡划分为多个较小的场景，以便艺术家和设计师可以在单个关卡上进行有效协作，同时最大限度地减少冲突风险。

请注意，在运行时，您的项目可以使用 `SceneManager.LoadSceneAsync` 传递 `LoadSceneMode.Additive` 参数 `mode` 来附加加载场景。

删除未使用的资源

注意与第三方插件和库捆绑在一起的任何未使用的资源。许多都包含嵌入式测试资产和脚本，如果您不删除它们，它们将成为您构建的一部分。去除原型设计中剩余的任何不需要的资源。

借助 Accelerate Solutions 的行业领先专业知识达到新的水平

Accelerate Solutions 专门帮助游戏工作室在多个用例中实现最雄心勃勃的目标，包括：提高性能和优化、游戏规划和技术设计、项目加速、提高玩家 KPI 和货币化，以及交付具有挑战性的端口和迁移。全球团队由 Unity 最资深的软件开发人员和技术美工人员组成，拥有 Unity 引擎、多人游戏、云、devops、AI/ML 和游戏设计方面的实践知识。

该团队的专业知识在于帮助您将游戏提升到一个新的水平，无论您处于游戏开发的哪个阶段。主要是，优化侧重于识别一般和特定的性能问题，例如帧速率、内存和二进制大小，以改善玩家体验和/或迭代时间。

提供的服务范围从咨询到完整的游戏开发。

— 咨询

在这些活动中，顾问将分析您的项目或工作流程，并向您的团队提供有关如何实现预期结果的指导和建议。

— 共同开发

Unity 开发人员和/或团队将与您的团队一起深入研究您的项目并实现预期的结果。

— 定制开发

对于这些项目，Accelerate Solutions 团队将指派一个内部 Unity 团队并与之合作，该团队将代表您领导和执行项目，并负责项目从开始到完成的整个过程。

— 完整的游戏开发

顾名思义，在这些项目中，Accelerate Solutions 会指派经验丰富的 Unity 游戏工作室团队，代表您领导和执行项目，并负责项目从开始到完成的整个过程。

要了解有关加速解决方案的更多信息，请[伸手](#)。

通过 Unity Integrated Success 消除障碍

从战略规划到不可预见的挑战，如果您需要个性化关注，请考虑[Unity 集成成功](#)。综合成功是我们针对您最复杂的项目的最完整的成功计划。获得见解、实践指导和优质技术支持，以确保您的项目取得成功。有保证的响应时间和优先的错误处理使您能够快速克服障碍。

Integrated Success 还允许您选择添加对 Unity 源代码的读取和修改访问权限。想要深入研究 Unity 源代码以将其改编并重用于其他应用程序的开发团队可以使用此访问权限。

通过项目审查优化您的游戏

项目评审是综合成功包的重要组成部分。了解如何在年度审核期间优化您的项目。高级工程师对您的工作进行分析，并针对您的目标提供见解和可行的建议。团队熟悉您的项目，然后使用各种分析工具来检测性能瓶颈，同时考虑现有需求和设计决策。他们还尝试找出可以优化性能的点，以提高速度、稳定性和效率。

对于构建时间短的架构良好的项目（模块化场景、大量使用 AssetBundles 等），他们将进行调整并重新分析以发现新问题。当团队无法立即解决问题时，他们会收集尽可能多的信息并在内部进行进一步调查，必要时咨询整个研发领域的专业开发人员。

尽管可交付成果可能会根据您的需求而有所不同，但我们会总结调查结果并在书面报告中提供建议。该团队的目标是通过帮助识别潜在的阻碍因素、评估风险、验证解决方案并确保遵循最佳实践，始终为您提供最大的价值。

合作伙伴关系经理 (PRM)

除了项目审查之外，Unity Integrated Success 还配备了合作伙伴关系经理 (PRM)，这是一位战略性 Unity 顾问，充当您的内部倡导者和团队的延伸，帮助您充分利用 Unity。他们保持清晰的沟通渠道，让您始终了解情况并努力实现您的目标。您的 PRM 为您提供所需的专业技术和运营专业知识，以预防问题并保持您的项目在启动之前和启动后顺利运行。

要了解有关我们的综合成功包、项目审查和 PRM 的更多信息，请[伸手](#)。

结论

您可以在以下位置找到更多优化技巧、最佳实践和新闻：[Unity 博客](#)和[Unity 社区论坛](#)，以及通过[统一学习](#)和 #unitytips 标签。

性能优化是一个需要仔细关注的广阔主题。了解移动硬件的运行方式及其局限性至关重要。为了找到满足您的设计要求的有效解决方案，您需要掌握 Unity 的类和组件、算法和数据结构以及平台的分析工具。

当然，一点点创造力在这里也有帮助。

更多资源

[创建 C# 风格指南：编写可扩展的更简洁的代码](#)帮助您开发风格指南，以帮助统一您创建更具凝聚力的代码库的方法。

[使用游戏编程模式升级您的代码](#)重点介绍使用 SOLID 原则和常见编程模式在 Unity 项目中创建可扩展游戏代码架构的最佳实践。

[使用 ScriptableObjects 在 Unity 中创建模块化游戏架构](#)提供专业开发人员在生产中部署 ScriptableObjects 的提示和技巧。其中包括展示如何将它们应用到特定设计模式以及如何避免常见陷阱的示例。

Unity 创作者的专业培训

Unity 专业培训为您提供技能和知识，使您能够在 Unity 中更高效地工作和高效协作。查找专为任何行业、任何技能水平、采用多种交付形式的专业人士设计的内容广泛的培训目录。

所有材料均由经验丰富的教学设计师与我们的工程师和产品团队合作创建。这意味着您始终会收到有关最新 Unity 技术的最新培训。

[了解更多有关 Unity 专业培训如何为您和您的团队提供支持。](#)



团结网