**Unity3D教程：代码优化整理**

Posted on 2013年07月05日 by U3d / [Unity3D 基础教程](http://www.unitymanual.com/category/manual/unity3d-%e5%9f%ba%e7%a1%80%e6%95%99%e7%a8%8b)/被围观 78 次

**减少固定增量时间**

将固定增量时间值设定在0.04-0.067区间（即，每秒15-25帧）。您可以通过Edit->Project Settings->Time来改变这个值。这样做降低了FixedUpdate函数被调用的频率以及物理引擎执行碰撞检测与刚体更新的频率。如果您使用了较低的固定增量时间，并且在主角身上使用了刚体部件，那么您可以启用插值办法来平滑刚体组件。

**减少GetComponent的调用**

使用 GetComponent或内置组件访问器会产生明显的开销。您可以通过一次获取组件的引用来避免开销，并将该引用分配给一个变量（有时称为“缓存”的引用）。例如，如果您使用如下的代码：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 1 | function Update () { |
| 2 |  |
| 3 | transform.Translate(0, 1, 0); |
| 4 |  |
| 5 | } |

通过下面的更改您将获得更好的性能：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | **var** myTransform : Transform; |
| 02 |  |
| 03 | function Awake () { |
| 04 |  |
| 05 | myTransform = transform; |
| 06 |  |
| 07 | } |
| 08 |  |
| 09 | function Update () { |
| 10 |  |
| 11 | myTransform.Translate(0, 1, 0); |
| 12 |  |
| 13 | } |
| 14 |  |

**避免分配内存**

您应该避免分配新对象，除非你真的需要，因为他们不再在使用时，会增加垃圾回收系统的开销。您可以经常重复使用数组和其他对象，而不是分配新的数组或对象。这样做好处则是尽量减少垃圾的回收工作。同时，在某些可能的情况下，您也可以使用结构（struct）来代替类（class）。这是因为，结构变量主要存放在栈区而非堆区。因为栈的分配较快，并且不调用垃圾回收操作，所以当结构变量比较小时可以提升程序的运行性能。但是当结构体较大时，虽然它仍可避免分配/回收的开销，而它由于“传值”操作也会导致单独的开销，实际上它可能比等效对象类的效率还要低。

**最小化GUI**

使用GUILayout 函数可以很方便地将GUI元素进行自动布局。然而，这种自动化自然也附带着一定的处理开销。您可以通过手动的GUI功能布局来避免这种开销。此外，您也可以设置一个脚本的useGUILayout变量为 false来完全禁用GUI布局：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 1 | function Awake () { |
| 2 |  |
| 3 | useGUILayout = **false**; |
| 4 |  |
| 5 | } |

**使用iOS脚本调用优化功能**

UnityEngine 命名空间中的函数的大多数是在 C/c + +中实现的。从Mono的脚本调用 C/C++函数也存在着一定的性能开销。您可以使用iOS脚本调用优化功能（菜单：Edit->Project Settings->Player）让每帧节省1-4毫秒。此设置的选项有：

Slow and Safe – Mono内部默认的处理异常的调用

Fast and Exceptions Unsupported –一个快速执行的Mono内部调用。不过，它并不支持异常，因此应谨慎使用。它对于不需要显式地处理异常（也不需要对异常进行处理）的应用程序来说，是一个理想的候选项。

**优化垃圾回收**

如上文所述，您应该尽量避免分配操作。但是，考虑到它们是不能完全杜绝的，所以我们提供两种方法来让您尽量减少它们在游戏运行时的使用：

如果堆比较小，则进行快速而频繁的垃圾回收

这一策略比较适合运行时间较长的游戏，其中帧率是否平滑过渡是主要的考虑因素。像这样的游戏通常会频繁地分配小块内存，但这些小块内存只是暂时地被使用。如果在iOS系统上使用该策略，那么一个典型的堆大小是大约 200 KB，这样在iPhone 3G设备上，垃圾回收操作将耗时大约 5毫秒。如果堆大小增加到1 MB时，该回收操作将耗时大约 7ms。因此，在普通帧的间隔期进行垃圾回收有时候是一个不错的选择。通常，这种做法会让回收操作执行的更加频繁（有些回收操作并不是严格必须进行的），但它们可以快速处理并且对游戏的影响很小：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 1 | **if** (Time.frameCount % 30 == 0) |
| 2 |  |
| 3 | { |
| 4 |  |
| 5 | System.GC.Collect(); |
| 6 |  |
| 7 | } |

但是，您应该小心地使用这种技术，并且通过检查Profiler来确保这种操作确实可以降低您游戏的垃圾回收时间。

如果堆比较大，则进行缓慢且不频繁的垃圾回收

这一策略适合于那些内存分配 (和回收）相对不频繁，并且可以在游戏停顿期间进行处理的游戏。如果堆足够大，但还没有大到被系统关掉的话，这种方法是比较适用的。但是，Mono运行时会尽可能地避免堆的自动扩大。因此，您需要通过在启动过程中预分配一些空间来手动扩展堆（ie，你实例化一个纯粹影响内存管理器分配的“无用”对象）：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | function Start() { |
| 02 |  |
| 03 | **var** tmp = new System.**Object**[1024]; |
| 04 |  |
| 05 | *// make allocations in smaller blocks to avoid them to be treated in a special way, which is designed for large blocks* |
| 06 |  |
| 07 | **for** (**var** i : **int** = 0; i < 1024; i++) |
| 08 |  |
| 09 | tmp[i] = new **byte**[1024]; |
| 10 |  |
| 11 | *// release reference* |
| 12 |  |
| 13 | tmp = **null**; |
| 14 |  |
| 15 | } |

游戏中的暂停是用来对堆内存进行回收，而一个足够大的堆应该不会在游戏的暂停与暂停之间被完全占满。所以，当这种游戏暂停发生时，您可以显式请求一次垃圾回收：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 1 | System.GC.Collect(); |

另外，您应该谨慎地使用这一策略并时刻关注Profiler的统计结果，而不是假定它已经达到了您想要的效果。