**Unity3D教程：性能优化**

Posted on 2013年06月04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 36 次

1、使用静态类型化

在JavaScript中，使用静态类型化（Static Typing）来代替动态类型化（Dynamic Typing）对性能的优化非常重要。Unity 使用一种叫做类型推导（Type Inference）的技术，自动把JavaScript转化为静态类型代码（Statically Typed Code），而不需要你做其他任何工作。

var foo = 5;像 foo 变量将会被自动推导为整数值。Unity 可以完成许多编译时优化（Compile TimeOptimization）的功能，并且不进行动态变量的耗时查找。这就是 Unity 的 JavaScript的执行速度比其他JavaScript快了20倍左右的原因之一。但是并不是所有变量都能进行类型推导，这时，Unity将回滚到动态类型化来处理它们。动态类型化时，在JavaScript中编写代码变得更加简单，但它的执行速度将会变慢。Unity3D教程手册 例如：

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

|  |  |
| --- | --- |
| 1 | function Start() |
| 2 |  |
| 3 | { |
| 4 |  |
| 5 | **var** foo = GetComponent(MyScript); |
| 6 |  |
| 7 | foo.DoSomething(); |
| 8 |  |
| 9 | } |

这儿的foo将进行动态类型化，因此，调用 DoSomething函数会耗时些。因为，编译器不知道foo的类型，它会去分析foo变量是否有 DoSomething函数，如果有，才进行调用。

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

|  |  |
| --- | --- |
| 1 | function Start() |
| 2 |  |
| 3 | { |
| 4 |  |
| 5 | **var** foo : MyScript = GetComponent(MyScript); |
| 6 |  |
| 7 | foo.DoSomething(); |
| 8 |  |
| 9 | } |

这里我们强制指定了foo的类型，这样会获得更高的性能。

2、使用#pragma strict

在脚本顶部增加#pragma strict语句，会让 Unity编译器在执行该脚本时关闭动态类型化支持，强制使用静态类型化。因此，如果变量的类型不确定，将导致编译错误。例如，下面的代码在编译时会产生错误：

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

|  |  |
| --- | --- |
| 01 | #pragma strict |
| 02 |  |
| 03 | function Start() |
| 04 |  |
| 05 | { |
| 06 |  |
| 07 | **var** foo = GetComponent(MyScript); |
| 08 |  |
| 09 | foo.DoSomething(); |
| 10 |  |
| 11 | } |

3、缓存组件查找

最优化的另一个方法是缓存组件， 但它需要编写额外的代码。如果脚本执行的次数很多，进行组件的缓存将获得更高的性能，这时编写额外的代码显然是值得的。在使用GetComponent函数或存取器变量（Accessor Variable）来访问一个组件时，Unit必须从游戏对象中查找相应的组件。这时， 我们可以使用私有变量来缓存一个指向该组件的引用，以便直接进行使用。Unity3D教程手册 因此，我们可以把：

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

|  |  |
| --- | --- |
| 1 | function Update() |
| 2 |  |
| 3 | { |
| 4 |  |
| 5 | transform.Translate(0, 0, 5); |
| 6 |  |
| 7 | } |
| 8 |  |

写成：

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

|  |  |
| --- | --- |
| 01 | **private** **var** myTransform : Transfrom; |
| 02 |  |
| 03 | function Awake() |
| 04 |  |
| 05 | { |
| 06 |  |
| 07 | myTransfrom = transform; |
| 08 |  |
| 09 | } |
| 10 |  |
| 11 | function Update() |
| 12 |  |
| 13 | { |
| 14 |  |
| 15 | myTransform.Translate(0, 0, 5); |
| 16 |  |
| 17 | } |
| 18 |  |

后面的代码将运行地更快，因为 Unity 不必每帧都在游戏对象中查找 transform 组件。这对于脚本组件也同样适用。

4、使用内建数组

内建数组的执行非常快，因此我们应该使用它。尽管ArrayList和 Array类的使用方式都比较简单，但它们的处理速度却有很大差别。内建数组都有固定的大小，通常事先我们都知道这个最大值。内建数组最大的优势在于它能够在一个紧凑的缓冲区中直接嵌入结构体，而不需要存储额外的类型信息。因此在缓存中迭代它时，将像在内存的一条线上进行处理，非常方便和快捷：

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

|  |  |
| --- | --- |
| 01 | **private** **var** positions : Vector3[]; |
| 02 |  |
| 03 | function Awake() |
| 04 |  |
| 05 | { |
| 06 |  |
| 07 | positions = new Vector3[100]; |
| 08 |  |
| 09 | **for**(**var** i=0; i<100; i++) |
| 10 |  |
| 11 | { |
| 12 |  |
| 13 | positions[i] = Vector3.zero; |
| 14 |  |
| 15 | } |
| 16 |  |
| 17 | } |
| 18 |  |

5、避免调用不必要的函数

最简单和有效的最优化在于减少不必要的工作。例如，当敌人离玩家很远时，我们可以让敌人静止不动，直到玩家走近它。一种较慢的处理如下：

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

|  |  |
| --- | --- |
| 01 | function Update() |
| 02 |  |
| 03 | { |
| 04 |  |
| 05 | *// Early out if the player is too far wary* |
| 06 |  |
| 07 | **if**(Vector3.Distance(transform.position, target.position) > 100) |
| 08 |  |
| 09 | **return**; |
| 10 |  |
| 11 | perform real work work ... |
| 12 |  |
| 13 | } |

这种处理的不足之处在于每帧都要执行 Update 函数。一种更好的解决方案是在玩家接近敌人时才启用脚本，有三种方式来进行实现：

(1)、使用 OnBecameVisible和OnBecameInvisible函数：它们的调用取决于渲染系统。当有摄像机能看到对象时，将调用 OnBecameVisible 函数；当没有摄像机能看到对象时，将调用OnBecameInvisible 函数。一般情况下这是有效的，但对于 Al 通常是无效的，因为一旦你把摄像机转开敌人时，他们可能会变得不可用。

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

|  |  |
| --- | --- |
| 01 | function OnBecameVisible() |
| 02 |  |
| 03 | { |
| 04 |  |
| 05 | enabled = **true**; |
| 06 |  |
| 07 | } |
| 08 |  |
| 09 | function OnBecameInvisible() |
| 10 |  |
| 11 | { |
| 12 |  |
| 13 | enabled = **false**; |
| 14 |  |
| 15 | } |
| 16 |  |

(2)、使用触发器：使用一个简单的球体触发器时，你可以根据球体的范围来调用

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

|  |  |
| --- | --- |
| 01 | OnTriggerEnter或OnTriggerExit函数。 |
| 02 |  |
| 03 | function OnTriggerEnter(c : Collider) |
| 04 |  |
| 05 | { |
| 06 |  |
| 07 | **if**(c.CompareTag(“Player”)) |
| 08 |  |
| 09 | enabled = ture; |
| 10 |  |
| 11 | } |
| 12 |  |
| 13 | function OnTriggerExit(c : Collider) |
| 14 |  |
| 15 | { |
| 16 |  |
| 17 | **if**(c.CompareTag(“Player”)) |
| 18 |  |
| 19 | enabled = **false**; |
| 20 |  |
| 21 | } |
| 22 |  |

(3)、使用协同程序：Update 函数会在每帧进行调用，我们完全可以使用协同程序来在每隔5秒钟检查一次距离，这将节省很多的资源。