**Unity官网针对ISO游戏的优化建议**

Posted on 2013年05月27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 20 次

尽量控制定点数量（注意所谓顶点不是建模时的顶点，而是引擎渲染时的顶点。例如，模型一个顶点如果设置了2个法向，那么对引擎来说就是2个顶点） ：

对Iphone3或更高设备，每帧渲染的顶点不超过4万个

对更早的设备，每帧渲染的顶点不超过1万个

场景所用的材质尽量减少。即使是不同的物件，也尽量采用同一个材质。

将固定的场景物件设置为静态（static）。

尽量使用PVRTC（这是Apple推荐的一种格式）纹理，不行的话，也尽量用16位纹理取代32位的。

不要在shader中使用multi-pass，通过用混合或者像素shader来合并不同的纹理以达到相同效果。

如果自行编写shader，尽量避免使用浮点数。下面是建议的类型：

fixed/lowp——适用于颜色，光照等 Unity3D教程手册

half/mediump——适用于纹理UV

float/highp——避免在像素shader里使用，可以在顶点shader里面用于计算顶点位置

在像素shader中尽量少用复杂的数学计算，例如幂函数、三角函数等。

没必要的话不要使用像素光照（Pixel Lights）

不要使用动态光照，用烘焙的lightmap

不要使用alpha测试，用alpha混合来代替

不要使用雾

大场景中使用Occlusion culling来削减不可见区域。关卡设计时考虑到实施 Occlusion culling 的便利性

使用sky box 来实现远景更多ISO优化相关教程：

[Unity3D教程：优化ISO](http://www.unitymanual.com/4372.html)

[Unity3D教程：ISO平台优化图形性能](http://www.unitymanual.com/3856.html)

[Unity3D教程：ISO系统性能与脚本的优化](http://www.unitymanual.com/3829.html)

**Unity3D教程：优化ISO**

Posted on 2013年04月30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 206 次

1.做减法

大量代码在Update()或FixedUpdate()中做处理，意味着无论代码的执行速度如何，都将在每次帧刷新的时候调用到如下代码：

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

|  |  |
| --- | --- |
| 01 | **public** **class** MyHoming : MonoBehaviour { |
| 02 |  |
| 03 | **public** **void** Update() { |
| 04 |  |
| 05 | *// expensive targetting and not so expensive trajectory calculati\*\*\*\*\* in here* |
| 06 |  |
| 07 | *// currently called every frame* |
| 08 |  |
| 09 | } |
| 10 |  |
| 11 | } |

但是大多数时候采用如下代码，看上去更复杂，但是只有在必要的时候才会执行代码。

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

|  |  |
| --- | --- |
| 01 | **public** **class** MyHoming : MonoBehaviour { |
| 02 |  |
| 03 | **public** **void** Update() { |
| 04 |  |
| 05 | *// expensive targetting and not so expensive trajectory calculati\*\*\*\*\* in here* |
| 06 |  |
| 07 | *// currently called every frame* |
| 08 |  |
| 09 | } |
| 10 |  |
| 11 | } |

2.强制指定垃圾回收的频率。不会改进你的fps，但是会降低内存的使用量。代码如下：

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

|  |  |
| --- | --- |
| 01 | **using** UnityEngine; |
| 02 |  |
| 03 | **class** GarbageCollectManager : MonoBehaviour { |
| 04 |  |
| 05 | **public** **int** frameFreq = 30; |
| 06 |  |
| 07 | **void** Update() { |
| 08 |  |
| 09 | **if** (Time.frameCount % frameFreq == 0) |
| 10 |  |
| 11 | System.GC.Collect(); |
| 12 |  |
| 13 | } |
| 14 |  |
| 15 | } |

3.三角形和绘图函数的调用计数

保持三角形低于7500，保持Draw Call低于20000，可能的话使用XCODE中的工具instruments来调试图形性能。

使用Strict：在你所有脚本的顶部使用#pragma ，脚本将是强类型的，避免写代码的时候错误的类型赋值。避免Object.Instantiate() 和 Object.Destroy()，Instantiating和Destroying 都不好，因为他们需要在创建和销毁对象的时候动态的分配内存，这样会短暂的影响到性能。

考虑的代替方案是使用他写的SpawnManager类，它在游戏一开始就初始化所有的对象，这样能在游戏载入的时候一同载入对象，从而让人感觉不到影响。

禁用对象的时候也保持在内存中，当需要他们的时候只需要在启用就行.

**Unity3D教程：ISO平台优化图形性能**

Posted on 2013年04月10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 307 次

[**Unity3D教程**](http://www.unitymanual.com/category/manual)：ISO平台优化图形性能

**Alpha 测试**

与桌面机不同，iOS 设备在进行alpha测试（或在像素着色器中使用丢弃和裁剪操作）时会造成很高的性能开销。所以，如果可以的话，应该尽可能地用alpha混合替换alpha测试。如果alpha测试不能避免，那么应该尽量将可见的 alpha测试像素总数降到最低。

**顶点性能**

一般来说，如果想在iPhone 3GS或更新的设备上每帧渲染不超过40,000可见点，那么对于一些配备 MBX GPU的旧设备（比如，原始的 iPhone，如 iPhone 3g和 iPod Touch第1和第2代）来说，应该保证每帧的渲染顶点在10000以下。

**光照性能**

逐像素的动态光照将对每个受影响的像素增加显着的计算开销，并可能导致物体会被渲染多次。为了避免这种情况的发生，应该避免对于任何单个物体都使用多个像素光照，并尽可能地使用方向光。需要注意的是像素光源是一个渲染模式（Render Mode）设置为重要（Important）的光源。逐像素的动态光照将对顶点变换增加显着的开销。所以，应该尽量避免任何给定的物体被多个光源同时照亮的情况。对于静态物体，采用烘焙光照方法则是更为有效的方法。

**优化模型几何**

当优化模型的几何数据时，有两个可以遵循的基本原则：不要使用不必要的三角形面片，尽量降低的 UV映射的接缝和硬边缘（ie，增加了一倍以上顶点）的数量，请注意，显卡所处理的实际顶点数目通常不等于一个 3D应用程序所报告的数目。建模工具通常显示的是模型的几何顶点数，ie，组成模型的不同角点的数目。但是，对于显卡来说，一些几何顶点往往需要根据渲染用途来拆分成两个或多个逻辑顶点。比如，一个顶点如果有多个法线、 UV坐标或顶点颜色，则必须对其进行拆分。因此，Unity的顶点计数总是比的 3D应用程序的顶点计数高很多。

**纹理压缩**

使用iOS自带的PVRT压缩格式可以有效地降低纹理的大小（这将导致更快的导入速度以及更小的内存占用），从而显着地提升渲染效率。压缩后的纹理所占用的内存带宽只相当于未压缩的 32位 RGBA 纹理的一小部分。在iOS Hardware Guide中，您可以找到的未压缩纹理与压缩纹理的性能比较。

某些图像经过PVRT 压缩后可能会在alpha通道中出现一些视觉瑕疵。在这种情况下，您需要直接在图像处理软件中调整 PVRT压缩参数。你可以安装PVR导出插件或直接使用Imagination Tech提供的PVRTexTool。生成的扩展名为.pvr的压缩图像文件，将直接被导入到Unity编辑器中，并且其指定的压缩参数将被保留。

如果PVRT压缩纹理无法给你足够好的视觉质量，或者你需要特别清晰的图片（比如用来作为GUI纹理），那么您应该考虑使用16位的纹理来代替RGBA纹理。因为这样做，你也可以减少一半的内存带宽。

**写出高性能的shader**

自从iPhone 3GS 以来，IOS 设备上的 Gpu就可以充分支持像素着色器和顶点着色器。但是，IOS设备上的性能还远远是没有达到PC机的程度，所以不应期望PC上的着色器可以端口不变地移植到IOS设备上。通常情况下，着色器需要进行一些优化来减少计算量和纹理读取，从而才能达到良好的性能。

**复杂的数学计算**

一些超越数学函数（如pow,exp,log,cos, sin,tan等）将增大GPU计算的负担，所以一个好的法则就是在每个片段中使用不多于一次的这种操作。在适当的情况下，可以考虑使用查找纹理来作为折中方法。

不建议自己写normalize,dot,inverse, sqrt等操作函数，如果使用内建函数，底层驱动将会为你生成更好质量的代码。切记，丢弃（discard）操作将会让你的片段运行缓慢。

**浮点数操作**

编写自定义着色器时，应该始终指定浮点变量的精度。为了获得最佳性能选择最小可浮动点格式至关重要。如果着色器是用GLSL ES来写的，那么浮点精度的指定如下所示：

highp–全32位浮点格式，适合顶点变换，但效率最低。

mediump –缩减的16位浮点格式，适合于纹理UV坐标，效率大致是highp的两倍。

lowp – 10位固定点格式，适合于颜色，光照计算和其他高性能操作，效率大致是highp的四倍。

如果着色器是用CG写的，或者它是一个表面着色器，那么它的精度指定如下所示：

float –与GLSL ES中的highp相似，效率最慢。

half -与GLSL ES中的mediump相似，效率大致是float的两倍。

fixed -与GLSL ES中的lowp相似，效率大致是float的四倍。

**硬件文档**

花费一些时间去学习Apple的硬件相关文档和best practices for writing shaders，但我们更加建议关注浮点型的精度问题。

**烘焙光照信息到光照贴图（Lightmap）**

通过Unity内建的Lightmapper，可以将场景的光照信息烘焙到纹理上。生成一个光照贴图要比仅仅在Unity的场景中放置一个光源来花费更长的时间，但是：它将运行更加快速（如果像素光源是2的话，光照贴图的运行速度将是动态光照的2-3倍）看上去效果更好，因为你可以烘焙全局光照明效果，同时光照贴图器还可以柔化（smooth）光照效果。

**共享材质**

如果当前帧中的许多物体都使用相同的材质，那么Unity iOS将会启动大量的内部优化，比如：避免设置各种不同的render state到OpenGL ES上。避免设置顶点和像素处理所需的不同参数的计算，批处理小型运动物体，来减少绘制调用，批处理开启静态属性的物体来减小绘制调用，所有这些优化将节省您宝贵的CPU周期。因此，进行纹理拼合使物体共用相同材质，这样的额外工作是会让你如愿以偿的。

**Unity3D教程：ISO系统性能与脚本的优化**

Posted on 2013年04月10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 222 次

**Unity3D**ISO系统性能与脚本的优化

**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）。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。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，代码如下：

function Update () {

transform.Translate(0, 1, 0);

}

通过下面的更改您将获得更好的性能：

var myTransform : Transform;

function Awake () {

myTransform = transform;

}

function Update () {

myTransform.Translate(0, 1, 0);

}

**避免分配内存**

应避免分配新对象，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。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，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。可以通过手动的GUI功能布局来避免这种开销。此外，也可以设置一个脚本的useGUILayout变量为 false来完全禁用GUI布局：

function Awake () {

useGUILayout = false;

}

**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。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，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：

if (Time.frameCount % 30 == 0)

{

System.GC.Collect();

}

但是，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）：

function Start() {

var tmp = new System.Object[1024];

// make allocations in smaller blocks to avoid them to be treated in a special way, which is designed for large blocks

for (var i : int = 0; i < 1024; i++)

tmp[i] = new byte[1024];

// release reference

tmp = null;

}

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，可以显式请求一次垃圾回收：

System.GC.Collect();

另外，应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想要的效果。