**Upgrading to Unity 4.0**

**更新到Unity4.0**

**GameObject active state**

**GameObject 激活状态**

Unity 4.0 changes how the active state of GameObjects is handled. GameObject’s active state is now inherited by child GameObjects, so that any GameObject which is inactive will also cause its children to be inactive. We believe that the new behavior makes more sense than the old one, and should have always been this way. Also, the upcoming new GUI system heavily depends on the new 4.0 behavior, and would not be possible without it. Unfortunately, this may require some work to fix existing projects to work with the new Unity 4.0 behavior, and here is the change:

Unity4.0更改了GameObject激活状态的使用方式。GameObject的激活状态现在被子GameObject继承，所以那些非激活状态的GameObject业火导致他的字节点不处于激活。我们相信这种新的机制比老的更为合理，而且就应该一直是这样的 。新的GUI系统很大程度上依赖于新的4.0机制，并且不可能脱离它运作。不幸的是，让现有的项目能够在新的Unity4.0机制下运行需要做一些工作，这里是一些修改：

**The old behavior:**

**旧的机制：**

* \* Whether a GameObject is active or not was defined by its **.active** property.

\* 通过**active**定义一个GameObject是否是激活状态。

* \* This could be queried and set by checking the **.active** property.
* \* 这可以通过**active**属性来查询或者赋值。
* \* A GameObject’s active state had no impact on the active state of child GameObjects. If you want to activate or deactivate a GameObject and all of its children, you needed to call **GameObject.SetActiveRecursively**.

\* 一个GameObject的激活状态不会影响其子节点GameObject的激活状态。如果你希望激活或者停用一个GameObject以及它的所有子节点，你应该调用**GameObject.SetActiveRecursively**.

* \* When using **SetActiveRecursively** on a GameObject, the previous active state of any child GameObject would be lost. When you deactivate and then activated a GameObject and all its children using **SetActiveRecursively**, any child which had been inactive before the call to **SetActiveRecursively**, would become active, and you had to manually keep track of the active state of children if you want to restore it to the way it was.
* \* 当对一个GameObject’使用**SetActiveRecursively** 时，所有的子GameObject之前的激活状态都会丢失。当你使用**SetActiveRecursively**来停用然后激活一个GameObject以及他的所有子节点时，那么所有在执行**SetActiveRecursively** 之前处于停用状态的子节点都会变成激活状态， 你需要手动的维护这些子节点的激活状态如果你希望能够恢复它们之前的状态的话。

\* Prefabs could not contain any active state, and were always active after prefab instantiation.

\* 预制中不包含任何的激活与否状态，在预制被实例化之后总是处于激活态。

**The new behavior:**

* \* Whether a GameObject is active or not is defined by its own **.activeSelf** property, and that of all of its parents. The GameObject is active if its own **.activeSelf** property and that of all of its parents is **true**. If any of them are **false**, the GameObject is inactive.

\* 一个GameObject是否是激活状态取决于它和它的父节点的**.activeSelf** 属性 。

* \* This can be queried using the **.activeInHierarchy** property.

\* 这可以通过**.activeInHierarchy属性来查询。**

* \* The **.activeSelf** state of a GameObject can be changed by calling **GameObject.SetActive**. When calling **SetActive (false)** on a previously active GameObject, this will deactivate the GameObject and all its children. When calling **SetActive (true)** on a previously inactive GameObject, this will activate the GameObject, if all its parents are active. Children will be activated when all their parents are active (i.e., when all their parents have **.activeSelf** set to **true**).

\* GameObject的**.activeSelf**状态可以通过调用**GameObject.SetActive**来修改。对一个激活状态的GameObject调用**SetActive (false)** ，这会停用这个GameObject以及它所有的子节点。对一个停用状态的GameObject执行**SetActive (true)**，这会激活这个GameObject如果它所有的父节点都处于激活状态的话。子节点在它所有父节点都处于激活状态时才会激活（比如当所有的父节点**.activeSelf** 属性都是 **true**）。

* \* This means that **SetActiveRecursively** is no longer needed, as active state is inherited from the parents. It also means that, when deactivating and activating part of a hierarchy by calling **SetActive**, the previous active state of any child GameObject will be preserved.
* \* 由于激活状态继承自父节点，这意味着不再需要**SetActiveRecursively**了。这也意味着，通过调用**SetActive**停用并且激活层级关系中的任一部分，那么之前每一个子GameObject的激活状态都会被保留

\* Prefabs can contain active state, which is preserved on prefab instantiation.

\* 预制可以包含激活与否的状态，这会在预制被实例化时保留。

**Example:**

**举例：**

You have three GameObjects, A, B and C, so that B and C are children of A.

你有3个GameObject，A,B和C，其中B和C是A的子节点。

* \* Deactivate C by calling **C.SetActive(false)**.
* \* 通过调用**C.SetActive(false)**.来停用C。
* \* Now, **A.activeInHierarchy == true**, **B.activeInHierarchy == true** and **C.activeInHierarchy == false**.
* \* 这时候**A.activeInHierarchy == true**, **B.activeInHierarchy == true** 并且 **C.activeInHierarchy == false**.
* \* Likewise, **A.activeSelf == true**, **B.activeSelf == true** and **C.activeSelf == false**.
* \* 同样的，**A.activeSelf == true**, **B.activeSelf == true** 并且 **C.activeSelf == false**.
* \* Now we deactivate the parent A by calling **A.SetActive(false)**.

\* 这时候我们通过调用**A.SetActive(false)**将父节点A停用。

* \* Now, **A.activeInHierarchy == false**, **B.activeInHierarchy == false** and **C.activeInHierarchy == false**.
* \* 这时候**A.activeInHierarchy == false**, **B.activeInHierarchy == false** 并且 **C.activeInHierarchy == false**.
* \* Likewise, **A.activeSelf == false**, **B.activeSelf == true** and **C.activeSelf == false**.
* \* 同样的，**A.activeSelf == false**, **B.activeSelf == true** 并且 **C.activeSelf == false**.
* \* Now we activate the parent A again by calling **A.SetActive(true)**.

\* 此时我们通过调用**A.SetActive(true)**.将父节点A激活。

* \* Now, we are back to **A.activeInHierarchy == true**, **B.activeInHierarchy == true** and **C.activeInHierarchy == false**.
* \* 此时我们回到了**A.activeInHierarchy == true**, **B.activeInHierarchy == true** 以及 **C.activeInHierarchy == false**.

\* Likewise, **A.activeSelf == true**, **B.activeSelf == true** and **C.activeSelf == false**.

\* 同样，**A.activeSelf == true**, **B.activeSelf == true** 并且 **C.activeSelf == false**.

**The new active state in the editor**

**在编辑器中的新的激活状态**

To visualize these changes, in the Unity 4.0 editor, any GameObject which is inactive (either because it’s own **.activeSelf** property is set to **false**, or that of one of it’s parents), will be greyed out in the hierarchy, and have a greyed out icon in the inspector. The GameObject’s own **.activeSelf** property is reflected by it’s active checkbox, which can be toggled regardless of parent state (but it will only activate the GameObject if all parents are active).

为了将这次改动可视化，在Unity4.0的编辑器中，所有处于停用状态的GameObject（无论是因为其自身还是它的父节点的**.activeSelf**被设置成了**false**）会被灰显，并且在inspector面板中灰显。GameObject自身的**.activeSelf**属性通过它的激活检查框（active checkbox）反馈，可以无视无节点的状态进行切换（但是仅会在其父节点都处于激活状态时才激活这个GameObject）。

**How this affects existing projects:**

**这会对现有的项目产生怎样的影响:**

* \* To make you aware of places in your code where this might affect you, the **GameObject.active** property and the **GameObject.SetActiveRecursively()** function have been deprecated.

\* 为了让你意识到那些在你代码中可能会影响你的地方，**GameObject.active**属性以及**GameObject.SetActiveRecursively()**函数都会标记为了不推荐（deprecated）。

* \* They are, however still functional. Reading the value of **GameObject.active** is equivalent to reading **GameObject.activeInHierarchy**, and setting **GameObject.active** is equivalent to calling **GameObject.SetActive()**. Calling **GameObject.SetActiveRecursively()** is equivalent to calling **GameObject.SetActive()** on the GameObject and all of it’s children.

\* 然而，他们仍然是功能性的。读取**GameObject.active** 的值和读取**GameObject.activeInHierarchy**是等价的，而对**GameObject.active**赋值和调用**GameObject.SetActive()**也是等价的。调用**GameObject.SetActiveRecursively()**和对GameObject以及其所有的子节点调用**GameObject.SetActive()**也是等价的。

* \* Exiting scenes from 3.5 are imported by setting the **selfActive** property of any GameObject in the scene to it’s previous **active** property.

\* 现有的3.5版本的场景在导入时会将GameObject的**selfActive**属性设置为之前**active**属性的值。

* \* As a result, any project imported from previous versions of Unity should still work as expected (with compiler warnings, though), as long as it does not rely on having active children of inactive GameObjects (which is no longer possible in Unity 4.0).

\* 结果是，所有从之前Unity版本中导入的工程都应该能如预期的运作（会带有编译警告），只要他不使用那些自身没有激活而子节点激活的GameObject（这在Unity4.0中是不可能实现的）

\* If your project relies on having active children of inactive GameObjects, you need to change your logic to a model which works in Unity 4.0.

\* 如果你的项目使用了那些自身没有激活而子节点激活的GameObject，你需要修改你的业务逻辑来使其能够在Unity4.0下工作。

**Changes to the asset processing pipeline**

**资源处理管道（asset processing pipeline）的改动**

During the development of 4.0 our asset import pipeline has changed in some significant ways internal in order to improve performance, memory usage and determinism. For the most part these changes does not have an impact on the user with one exception: Objects in assets are not made persistent until the very end of the import pipeline and any previously imported version of an assets will be completely replaced.

在4.0的开发中，我们的资源导入管道（asset import pipeline）发生了一些重大的改动以提高性能，内存使用以及确定性。大部分情况下这些改动不会影响用户，除了：资源包中的Objects是不持久的，直到导入过程的结束，任何之前导入的资源会被完全的替换。

The first part means that during post processing you cannot get the correct references to objects in the asset and the second part means that if you use the references to a previously imported version of the asset during post processing do store modification those modifications will be lost.

第一部分意味着你后期处理阶段你不能获取资源包中objects正确的引用，第二部分意味着你在后期处理阶段引用了之前导入的资源包来作一些修改，那么这些修改是会丢失的。

**Example of references being lost because they are not persistent yet**

**关于不持久性导致的引用丢失例子**

Consider this small example:

想想这个小例子:

public class ModelPostprocessor : AssetPostprocessor

{

public void OnPostprocessModel(GameObject go)

{

PrefabUtility.CreatePrefab("Prefabs/" + go.name, go);

}

}

In Unity 3.5 this would create a prefab with all the correct references to the meshes and so on because all the meshes would already have been made persistent, but since this is not the case in Unity 4.0 the same post processor will create a prefab where all the references to the meshes are gone, simply because Unity 4.0 does not yet know how to resolve the references to objects in the original model prefab. To correctly copy a modelprefab in to prefab you should use **OnPostProcessAllAssets** to go through all imported assets, find the modelprefab and create new prefabs as above.

在Unity3.5中这将会创建一个拥有所有网格正确引用的预制，因为所有的网格都已经被持久化了，但是这在Unity4.0中不一样，同样的后期处理程序会创建一个丢失了所有网格引用的预制，这仅仅是因为Unity4.0不知道怎么解决在原来的预制中对objects的引用。要正确的将模板预制（modelprefab）拷贝一个预制，你应该使用**OnPostProcessAllAssets**来实现在所有资源导入后找模板预制（modelprefab)并且像上面那样创建新的预制。

**Example of references to previously imported assets being discarded**

**之前加载的资源包的引用被丢弃的例子**

The second example is a little more complex but is actually a use case we have seen in 3.5 that broke in 4.0. Here is a simple **ScriptableObject** with a references to a mesh.

第二个例子有一点复杂，但是这确实是我们在3.5中看到而在4.0中没有的用例。这里是一个简单的带有网格引用的**ScriptableObject**

public class Referencer : ScriptableObject

{

public Mesh myMesh;

}

We use this **ScriptableObject** to create an asset with references to a mesh inside a model, then in our post processor we take that reference and give it a different name, the end result being that when we have reimported the model the name of the mesh will be what the post processor determines.

我们使用这个**ScriptableObject** 来创建一个引用了某个模型中的网格的资源包，那么在后期处理程序中获取那个引用，然后给予其一个不同的名字，最终的结果是，当我们重新导入一个模型时，网格的名字是由后期处理程序决定的。

public class Postprocess : AssetPostprocessor

{

public void OnPostprocessModel(GameObject go)

{

Referencer myRef = (Referencer)AssetDatabase.LoadAssetAtPath("Assets/MyRef.asset", typeof(Referencer));

myRef.myMesh.name = "AwesomeMesh";

}

}

This worked fine in Unity 3.5 but in Unity 4.0 the already imported model will be completely replaced, so changing the name of the mesh from a previous import will have no effect. The Solution here is to find the mesh by some other means and change its name. What is most important to note is that in Unity 4.0 you should ONLY modify the given input to the post processor and not rely on the previously imported version of the same asset.

这在Unity 3.5中能正常运作但是在Unity 4.0中，已经导入的模型会被完全替换，所以在导入前修改网格的名字没有任何效果。这里的解决方案是用其他的方式找到网格并且修改它的名字。最重要的是在Unity 4.0中你赢只修改在后期处理程序中输入的内容并且不要依赖同一个资源包中之前导入的版本。

**Mesh Read/Write option**

**网格的读/写选项**

Unity 4.0 adds a “Read/Write Enabled” option in [Mesh](http://docs.unity3d.com/540/Documentation/Manual/class-FBXImporter.html) import settings. When this option is turned off, it saves memory since Unity can unload a copy of mesh data in the game.

Unity 4.0 在网格导入设置中增加了一个“Read/Write Enabled” 的选项。当这个选项关闭时， Unity就可以在游戏中卸载网格数据的拷贝，这会节省内存。

However, if you are scaling or instantiating meshes at runtime with a non-uniform scale, you may have to enable “Read/Write Enabled” in their import settings. The reason is that non-uniform scaling requires the mesh data to be kept in memory. Normally we detect this at build time, but when meshes are scaled or instantiated at runtime you need to set this manually. Otherwise they might not be rendered in game builds correctly.

然而，如果你在运行时使用一个不规范的比例去缩放或者实例化网格，你就需要在设置中激活“Read/Write Enabled”.原因是不规则的缩放需要在内存中维护一份网格数据。通常情况下我们在生成时会发现这些，但是当网格在运行时缩放或者实例化，你就需要手动去控制。此外，他们将不会在生成的终端中正确渲染。

**Mesh optimization**

**网格优化**

The Model Importer in Unity 4.0 has become better at mesh optimization. The “Mesh Optimization” checkbox in the Model Importer in Unity 4.0 is now enabled by default, and will reorder the vertices in your Mesh for optimal performance. You may have some post-processing code or effects in your project which depend on the vertex order of your meshes, and these might be broken by this change. In that case, turn off “Mesh Optimization” in the Mesh importer. Especially, if you are using the SkinnedCloth component, mesh optimization will cause your vertex weight mapping to change. So if you are using SkinnedCloth in a project imported from 3.5, you need to turn off “Mesh Optimization” for the affected meshes, or reconfigure your vertex weights to match the new vertex order.

Unity 4.0的模型导入器在网格优化上做的更好。“Mesh Optimization”选择框在Unity4.0的模型导入器中是默认选中的，这会通过对模型中的顶点重新排序来优化性能。你可以会有一些基于网格顶点排序的效果或者后期处理代码，都会被这次改动所破坏。在这种情况下，就在网格导入器中关闭“Mesh Optimization”.尤其是，当你正在使用SkinnedCloth组件，网格优化会导致你的顶点权重映射产生修改。所以当你在项目中使用了3.5版本导入的 SkinnedCloth时，你需要关闭相关网格的“Mesh Optimization”,或者重新配置你的顶点权重以匹配新的顶点顺序。

**Mobile input**

**移动输入**

With Unity 4.0 mobile sensor input got better alignment between platforms, which means you can write less code when handling typical input on mobile platforms. Now acceleration and gyro input will follow screen orientation in the same way both on iOS and Android platforms. To take advantage of this change you should refactor your input code and remove platform and screen orientation specific code when handling acceleration and gyro input. You still can get old behavior on iOS by setting **Input.compensateSensors** to false.

在Unity4.0中，移动设备的传感器输入在不同设备间的兼容性更好了，这意味着你可以在处理移动平台的典型输入时少写一些代码。现在加速计以及陀螺仪输入在iOS和Android平台上会以同样的方式跟随屏幕的方向。所以，利用这次改动带来的好处去重构你的代码，移除那些使用加速计和陀螺仪时对屏幕方向和平台做特殊处理的代码。你依然可以在iOS上通过将**Input.compensateSensors**设置为false来获取旧的机制