[图文教程] 使用Unity3D的50个技巧：Unity3D最佳实践

neil3d

威望 0 点

义气 5 点

打招呼 发消息

发表于 2014-9-5 07:39:08 | 只看该作者 楼主

刚开始学习Unity3D时间不长，在看各种资料。除了官方的手册以外，其他人的经验也是非常有益的。偶尔看到老外这篇文章，觉得还不错，于是翻译过来和大家共享。原文地址：http://devmag.org.za/2012/07/12/50-tips-for-working-with-unity-best-practices/，下面是译文。

欢迎转载，请注明出处：燕良@游戏开发，http://blog.csdn.net/neil3d/article/details/38534809。另外，欢迎各路高手加入我的QQ群：264656505，切磋交流技术。

关于这些技巧这些技巧不可能适用于每个项目。

- 这些是基于我的一些项目经验，项目团队的规模从3人到20人不等；
- 框架结构的可重用性、清晰程度是有代价的——团队的规模和项目的规模决定你要在这个上面付出多少；
- 很多技巧是品味的问题（这里所列的所有技巧，可能有同样好的技术替代方案）；
- 一些技巧可能是对传统的Unity开发的一个冲击。例如，使用prefab替代对象实例并不是一个传统的Unity风格，并且这样做的代价还挺高的（需要很多的preffab）。也许这些看起来有些疯狂，但是在我看来是值得的。

流程

1、避免**Assets**分支

所有的Asset都应该只有一个唯一的版本。如果你真的需要一个分支版本的Prefab、Scene或是Mesh，那你要制定一个非常清晰的流程，来确定哪个是正确的版本。错误的分支应该起一个特别的名字，例如双下划线前缀：__MainScene_Backup。Prefab版本分支需要一个特别的流程来保证安全（详见Prefabs一节）。

2、如果你在使用版本控制的话，每个团队成员都应该保有一个项目的**Second Copy**用来测试修改之后，Second Copy和Clean Copy都应该被更新和测试。大家都不要修改自己的Clean Copy。这对于测试Asset丢失特别有用。

3、考虑使用外部的关卡编辑工具Unity不是一个完美的关卡编辑器。例如，我们使用TuDee来创建3D Tile-Based的游戏，这使我们可以获得对Tile友好的工具的益处（网格约束，90度倍数的旋转，2D视图，快速Tile选择等）。从一个XML文件来实例化Prefab也很简单。详见Guerrilla Tool Development。

4、考虑把关卡保存为**XML**，而非**scene**这是一种很奇妙的技术：

- 它可以让你不必每个场景都设置一遍；
- 他可以加载的更快（如果大多数对象都是在场景之间共享的）。
- 它让场景的版本合并变的简单（就算是Unity的新的文本格式的Scene，也由于数据太多，而让版本合并变的不切实际）。
- 它可以使得在关卡之间保持数据更简便。

你仍就可以使用Unity作为关卡编辑器（尽管你用不着了）。你需要写一些你的数据的序列化和反序列化的代码，并实现在编辑器和游戏运行时加载关卡、在编辑器中保存关卡。你可能需要模仿Unity的ID系统来维护对象之间的引用关系。

5、考虑编写通用的自定义**Inspector**代码实现自定义的Inspector是很直截了当的，但是Unity的系统有很多的缺点：

- 它不支持从继承中获益；
- 它不允许定义字段级别的Inspector组件，而只能是class类型级别。举个例子，如果没有游戏对象都有一个ScomeCoolType字段，而你想在Inspector中使用不同的渲染，那么你必须为你的所有class写Inspector代码。

你可以通过从根本上重新实现Inspector系统来处理这些问题。通过一些反射机制的小技巧，他并不像看上去那么看，文章底部（日后另作翻译）将提供更多的实现细节。

场景组织**6**、使用命名的空**Game Object**来做场景目录仔细的组织场景，就可以方便的找到任何对象。

7、把控制对象和场景目录（空**Game Objec**）放在原点（**0**，**0**，**0**）如果位置对于这个对象不重要，那么就把他放到原点。这样你就不会遇到处理Local Space和World Space的麻烦，代码也会更简洁。

8、尽量减少使用**GUI**组件的**offset**通常应该由控件的Layout父对象来控制Offset；它们不应该依赖它们的爷爷节点的位置。位移不应该互相抵消来达到正确显示的目的。做基本上要防止了下列情况的发生：

父容器被放到了（100，-50），而字节点应该在（10，10），所以把他放到（90，60）[父节点的相对位置]。

这种错误通常放生在容器不可见时。

9、把世界的地面放在**Y=0**这样可以更方便的把对象放到地面上，并且在游戏逻辑中，可以把世界作为2D空间来处理（如果合适的话），例如AI和物理模拟。

10、使游戏可以从每个**Scene**启动这将大大的降低测试的时间。为了达到所有场景可运行，你需要做两件事：

首先，如果需要前面场景运行产生的一些数据，那么要模拟出它们。

其次，生成在场景切换时必要保存的对象，可以是这样：

```
myObject = FindMyObjectInScene(); if (myObjet == null){  myObject = SpawnMyObject();}
```

美术**11**、把角色和地面物体的中心点**(Pivot)**放在底部，不要放在中间这可以使你方便的把角色或者其他对象精确的放到地板上。如果合适的话，它也可能使得游戏逻辑、AI、甚至是物理使用2D逻辑来表现3D。

12、统一所有的模型的面朝向（**Z**轴正向或者反向）对于所有具有面朝向的对象（例如角色）都应该遵守这一条。在统一面朝向的前提下，很多算法可以简化。

13、在开始就把**Scale**搞正确请美术把所有导入的缩放系数设置为1，并且把他们的Transform的Scale设置为1,1,1。可以使用一个参考对象（一个Unity的Cube）来做缩放比较。为你的游戏选择一个世界的单位系数，然后坚持使用它。
14、为**GUI**组件或者手动创建的粒子制作一个两个面的平面模型设置这个平面面朝向Z轴正向，可能简化Billboard和GUI创建。
15、制作并使用测试资源

- 为SkyBox创建带文字的方形贴图；
- 一个网格（Grid）；
- 为Shader测试使用各种颜色的平面：白色，黑色，50%灰度，红，绿，蓝，紫，黄，青；
- 为Shader测试使用渐进色：黑到白，红到绿，红到蓝，绿到蓝；
- 黑白格子；
- 平滑的或者粗糙的法线贴图；
- 一套用来快速搭建场景的灯光（使用Prefa）；

Prefabs16、所有东西都使用**Prefab**只有场景中的"目录"对象不使用Prefab。甚至是那些只使用一次的唯一对象也应该使用Prefab。这样可以在不动用场景的情况下，轻松修改他们。（一个额外的好处是，当你使用EZGUI时，这可以用来创建稳

定的Sprite Atlases）
17、对于特例使用单独的**Prefab**，而不要使用特殊的实例对象如果你有两种敌人的类型，并且只是属性有区别，那么为不同的属性分别创建Prefab，然后链接他们。这可以：

- 在同一个地方修改所有类型
- 在不动用场景的情况下进行修改

如果你有很多敌人的类型，那么也不要在编辑器中使用特殊的实例。一种可选的方案是程序化处理它们，或者为所有敌人使用一个核心的文件/Prefab。使用一个下拉列表来创建不同的敌人，或者根据敌人的位置、玩家的进度来计算。
18、在**Prefab**之间链接，而不要链接实例对象当Prefab放置到场景中时，它们的链接关系是被维护的，而实例的链接关系不被维护。尽可能的使用Prefab之间的链接可以减少场景创建的操作，并且减少场景的修改。
19、如果可能，自动在实例对象之间产生链接关系如果你确实需要在实例之间链接，那么应该在程序代码中去创建。例如，Player对象在Start时需要把自己注册到GameManager，或者GameManager可以在Start时去查找Player对象。
对于需要添加脚本的Prefab，不要用Mesh作为根节点。当你需要从Mesh创建一个Prefab时，首先创建一个空的GameObject作为父对象，并用来做根节点。把脚本放到根节点上，而不要放到Mesh节点上。使用这种方法，当你替换Mesh时，就不会丢失所有你在Inspector中设置的值了。
使用互相链接的Prefab来实现Prefab嵌套。Unity并不支持Prefab的嵌套，在团队合作中第三方的实现方案可能是危险的，因为嵌套的Prefab之间的关系是不明确的。
20、使用安全的流程来处理**Prefab**分支我们用一个名为Player的Prefab来讲解这个过程。
用下面这个流程来修改Player：

1. 复制Player Prefab；
2. 把复制出来的Prefab重命名为__Player_Backup；
3. 修改Player Prefab；
4. 测试一切工作正常，删除__Player_Backup；

不要把新复制的命名为Player_New，然后修改它。
有些情况可能更复杂一些。例如，有些修改可能涉及到两个人，上述过程有可能使得场景无法工作，而所有人必须停下来等他们修改完毕。如果修改能够很快完成，那么还用上面这个流程就可以。如果修改需要花很长时间，则可以使用下面的流程：

1. 第一个人：
    1. 复制Player Prefab；
    2. 把它重命名为__Player_WithNewFeature或者__Player_ForPerson2；
    3. 在复制的对象上做修改，然后提交给第二个人；
2. 第二个人：
    1. 在新的Prefab上做修改；
    2. 复制Player Prefab，并命名为__Player_Backup；
    3. 把__Player_WithNewFeature拖放到场景中，创建它的实例；
    4. 把这个实例拖放到原始的Player Prefab中；
    5. 如果一切工作正常，则可使删除__Player_Backup和__Player_WithNewFeature;

扩展和**MonoBehaviourBase21**、扩展一个自己的**Mono Behaviour**基类，然后自己的所有组件都从它派生这可以使你方便的实现一些通用函数，例如类型安全的Invoke，或者是一些更复杂的调用（例如random等等）。
22、为**Invoke, StartCoroutine and Instantiate** 定义安全调用方法定义一个委托任务（delegate Task），用它来定义

需要调用的方法，而不要使用字符串属性方法名称，例如：
public void Invoke(Task task, float time){ Invoke(task.Method.Name, time);}
23、为共享接口的组件扩展有些时候把获得组件、查找对象实现在一个组件的接口中会很方便。
下面这种实现方案使用了typeof，而不是泛型版本的函数。泛型函数无法在接口上工作，而typeof可以。下面这种方法把泛型方法整洁的包装起来。
//Defined in the common base class for all mono behaviourspublic I GetInterfaceComponent<I>() where I : class{ return GetComponent(typeof(I)) as I;} public static List<I> FindObjectsOfInterface<I>() where I : class{ MonoBehaviour[] monoBehaviours = FindObjectsOfType<MonoBehaviour>(); List<I> list = new List<I>(); foreach(MonoBehaviour behaviour in monoBehaviours) { I component = behaviour.GetComponent(typeof(I)) as I; if(component != null)

{ list.Add(component); } } return list;}

24、使用扩展来让代码书写更便捷例如：

public static class CSTransform { public static void SetX(this Transform transform, float x) { Vector3 newPosition = new Vector3(x, transform.position.y, transform.position.z); transform.position = newPosition; } ...}

25、使用防御性的**GetComponent()**有些时候强制性组件依赖（通过RequiredComponent）会让人蛋疼。例如，很难在Inspector中修改组件（即使他们有同样的基类）。下面是一种替代方案，当一个必要的组件没有找到时，输出一条错误信息。

public static T GetSafeComponent<T>(this GameObject obj) where T : MonoBehaviour{ T component = obj.GetComponent<T>(); if(component == null) { Debug.LogError("Expected to find component of type " + typeof(T) + " but found none", obj); } return component;}

风格**26**、避免对同一件事使用不同的处理风格在很多情况下，某件事并不只有一个惯用手法。在这种情况下，在项目中明确选择其中的一个来使用。下面是原因：

- 一些做法并不能很好的一起协作。使用一个，能强制统一设计方向，并明确指出不是其他做法所指的方向；
- 团队成员使用统一的风格，可能方便大家互相的理解。他使得整体结构和代码都更容易理解。这也可以减少错误；

几组风格的例子：

- 协程与状态机（Coroutines vs. state machines）；
- 嵌套的Prefab、互相链接的Prefab、超级Prefab（Nested prefabs vs. linked prefabs vs. God prefabs）；
- 数据分离的策略；
- 在2D游戏的使用Sprite的方法；
- Prefab的结构；
- 对象生成策略；
- 定位对象的方法：使用类型、名称、层、引用关系；
- 对象分组的方法：使用类型、名称、层、引用数组；
- 找到一组对象，还是让它们自己来注册；
- 控制执行次序（使用Unity的执行次序设置，还是使用Awake/Start/Update/LateUpdate，还是使用纯手动的方法，或者是次序无关的架构）；
- 在游戏中使用鼠标选择对象/位置/目标：SelectionManager或者是对象自主管理；
- 在场景变换时保存数据：通过PlayerPrefs，或者是在新场景加载时不要销毁的对象；
- 组合动画的方法：混合、叠加、分层；

时间**27**、维护一个自己的**Time**类，可以使游戏暂停更容易实现做一个“Time.DeltaTime”和""Time.TimeSinceLevelLoad"的包装，用来实现暂停和游戏速度缩放。这使用起来略显麻烦，但是当对象运行在不同的时钟速率下的时候就方便多了（例如界面动画和游戏内动画）。

生成对象**28**、不要让游戏运行时生成的对象搞乱场景层次结构在游戏运行时，为动态生成的对象设置好它们的父对象，可以让你更方便的查找。你可以使用一个空的对象，或者一个没有行为的单件来简化代码中的访问。可以给这个对象命名为“DynamicObjects”。

类设计**29**、使用单件（**Singleton**）模式

从下面这个类派生的所有类，将自动获得单件功能：

public class Singleton<T> : MonoBehaviour where T : MonoBehaviour{ protected static T instance; /** Returns the instance of this singleton. */ public static T Instance { get { if(instance == null) { instance = (T) FindObjectOfType(typeof(T)); if (instance == null) { Debug.LogError("An instance of " + typeof(T) + " is needed in the scene, but there is none."); } } return instance; } }单件可以作为一些管理器，例如ParticleManager或者AudioManager亦或者GUIManager。

- 对于那些非唯一的prefab实例使用单件管理器（例如Player）。不要为了坚持这条原则把类的层次关系复杂化，宁愿在你的GameManager（或其他合适的管理器中）中持有一个它们的引用。
- 对于外部经常使用的共有变量和方法定义为static，这样你可以这样简便的书写“GameManager.Player”，而不用写成“GameManager.Instance.player”。

30、在组件中不要使用**public**成员变量，除非它需要在**inspector**中调节

除非需要设计师（策划or美术）去调节的变量，特别是它不能明确表明自己是做什么的变量，不要声明为public。如果在这些特殊情况下，无法避免，则可使用两个甚至四个下划线来表明不要从外部调节它，例如：

public float __aVariable;

31、把界面和游戏逻辑分开这一条本质上就是指的MVC模式。

所有的输入控制器，只负责向相应的组件发送命令，让它们知道控制器被调用了。举一个控制器逻辑的例子，一个控制器根据玩家的状态来决定发送哪个命令。但是这样并不好（例如，如果你添加了多个控制器，那将会导致逻辑重复）。相反的，玩家对象应该根据当前状态（例如减速、惊恐）来设置当前的速度，并根据当前的面朝向来计算如何向前移动。控制器只负责做他们自己状态相关的事情，控制器不改变玩家的状态，因此控制前甚至可以根本不知道玩家的状态。另外一个例子，切换武器。正确的方法是，玩家有一个函数：“SwitchWeapon(Weapon newWeapon)”供GUI调用。GUI不应该维护所有对象的Transform和他们之间的父子关系。

所有界面相关的组件，只负责维护和处理他们自己状态相关的数据。例如，显示一个地图，GUI可以根据玩家的位移计算地

图的显示。但是，这是游戏状态数据，它不属于GUI。GUI只是显示游戏状态数据，这些数据应该在其他地方维护。地图数据也应该在其他地方维护（例如GameManager）。

游戏玩法对象不应该关心GUI。有一个例外是处理游戏暂停（可能是通过控制Time.timeScale，其实这并不是个好主意）。游戏玩法对象应该知道游戏是否暂停。但是，这就是全部了。另外，不要把GUI组件挂到游戏玩法对象上。

这么说吧，如果你把所有的GUI类都删了，游戏应该可以正确编译。

你还应该达到：在不需要重写游戏逻辑的前提下，重写GUI和输入控制。

32、分离状态控制和簿记变量簿记变量只是为了使用起来方便或者提高查找速度，并且可以根据状态控制来覆盖。将两者分离可以简化：

- 保存游戏状态
- 调试游戏状态

实现方法之一是为每个游戏逻辑定义一个"SaveData"类，例如：
[Serializable]PlayerSaveData{ public float health; //public for serialisation, not exposed in inspector} Player{ //... bookkeeping variables //Don't expose state in inspector. State is not tweakable. private PlayerSaveData playerSaveData; }
33、分离特殊的配置假设我们有两个敌人，它们使用同一个Mesh，但是有不同的属性设置（例如不同的力量、不同的速度等等）。有很多方法来分离数据。下面是我比较喜欢的一种，特别是对于对象生成或者游戏存档时，会很好用。（属性设置不是状态数据，而是配置数据，所以我们不需要存档他们。当对象加载或者生成是，属性设置会自动加载。）

- 为每一个游戏逻辑类定义一个模板类。例如，对于敌人，我们来一个"EnemyTemplate"，所有的属性设置变量都保存在这个类中。
- 在游戏逻辑的类中，定义一个上述模板类型的变量。
- 制作一个敌人的Prefab，以及两个模板的Prefab："WeakEnemyTemplate"和"StrongEnemyTemplate"。
- 在加载或者生成对象是，把模板变量正确的复制。

这种方法可能有点复杂（在一些情况下，可能不需要这样）。

举个例子，最好使用泛型，我们可以这样定义我们的类：
public class BaseTemplate{ ...} public class ActorTemplate : BaseTemplate{ ...} public class Entity<EntityTemplateType> where EntityTemplateType : BaseTemplate{ EntityTemplateType template; ...} public class Actor : Entity <ActorTemplate>{ ...}
34、除了显示用的文本，不要使用字符串特别是不要用字符串作为对象或者prefab等等的ID标识。一个很遗憾的例外是动画系统，需要使用字符串来访问相应的动画。
35、避免使用**public**的数组举例说明，不要定义一个武器的数组，一个子弹的数组，一个粒子的数组，这样你的代码看起来像这样：
public void SelectWeapon(int index){ currentWeaponIndex = index; Player.SwitchWeapon(weapons[currentWeapon]);} public void Shoot(){ Fire(bullets[currentWeapon]); FireParticles(particles[currentWeapon]); }
这在代码中还不是什么大问题，但是在Inspector中设置他们的值的时候，就很难不犯错了。

我们可以定义一个类，来封装这三个变量，然后使用一个它的实例数组：
[Serializable]public class Weapon{ public GameObject prefab; public ParticleSystem particles; public Bullet bullet;} 这样代码看起来很整洁，但是更重要的是，在Inspector中设置时就不容易犯错了。

36、在结构中避免使用数组举个例子，一个玩家可以有三种攻击形式，每种使用当前的武器，并发射不同的子弹、产生不同的行为。

三个子弹作为一个数组，并像下面这样组织逻辑：
oid FireAttack(){ /// behaviour Fire(bullets[0]);} public void IceAttack(){ /// behaviour Fire(bullets[1]);} public ndAttack(){ /// behaviour Fire(bullets[2]);} 使用枚举值可以让代码看起来更好一点：

lic void WindAttack(){ /// behaviour Fire(bullets[WeaponType.Wind]);}
是这对Inspector一点也不好。

好使用单独的变量，并且起一个好的变量名，能够代表他们的内容的含义。使用下面这个类会更整洁。
rializable]public class Bullets{ public Bullet FireBullet; public Bullet IceBullet; public Bullet WindBullet;}这里假设没有其他ire、Ice、Wind的数据。

、把数据组织到可序列化的类中，可以让**inspector**更整洁有些对象有一大堆可调节的变量，这种情况下在Inspector中找到某个变量简直就成了噩梦。为了简化这种情况，可以使用一下的步骤：

- 把这些变量分组定义到不同的类中，并让它们声明为public和serializable；
- 在一个主要的类中，把上述类的实例定义为public成员变量；
- 不用在Awake或者Start中初始化这些变量，因为Unity会处理好它们；
- 你可以定义它们的默认值；

这可以把变量分组到Inspector的分组页签中，方便管理。

[Serializable]public class MovementProperties //Not a MonoBehaviour!{ public float movementSpeed; public float turnSpeed = 1; //default provided} public class HealthProperties //Not a MonoBehaviour!{ public float maxHealth; public float regenerationRate;} public class Player : MonoBehaviour{ public MovementProperties movementProeprties; public HealthProperties healthProeprties;}

HealthProperties healthProperties;}

文本**38**、如果你有很多的剧情文本，那么把他们放到一个文件里面。不要把他们放到Inspector的字段中去编辑。这些需要做到不打开Unity，也不用保存Scene就可以方便的修改。
39、如果你计划实现本地化，那么把你的字符串分离到一个统一的位置。有很多种方法来实现这点。例如，定义一个文本Class，为每个字符串定义一个public的字符串字段，并把他们的默认值设为英文。其他的语言定义为子类，然后重新初始化这些字段为相应的语言的值。
另外一种更好的技术（适用于文本很大或者支持的语言数量众多），可以读取几个单独的表单，然后提供一些逻辑，根据所选择的语言来选取正确的字符串。

测试与调试**40**、实现一个图形化的**Log**用来调试物理、动画和**AI**。这可以显著的加速调试工作。详见这里。
41、实现一个**HTML**的**Log**。在很多情况下，日志是非常有用的。拥有一个便于分析的Log（颜色编码、有多个视图、记录屏幕截图等）可以使基于Log的调试变动愉悦。详见这里。
42、实现一个你自己的帧速率计算器。没有人知道Unity的FPS计算器在做什么，但是肯定不是计算帧速率。实现一个你自己的，让数字符合直觉并可视化。
43、实现一个截屏的快捷键。很多BUG是图形化的，如果你有一个截图，就很容易报告它。一个理想的系统，应该在PlayerPrefes中保存一个计数，并根据这个计数，使得所有成功保存的截屏文件都不被覆盖掉。截屏文件应该保存在工程文件夹之外，这可以防止人们不小心把它提交到版本库中。
44、实现一个打印玩家坐标的快捷键。这可以在汇报位置相关的BUG时明确它发生在世界中的什么位置，这可以让Debug容易一些。
45、实现一些**Debug**选项，用来方便测试。一些例子：

- 解锁所有道具；
- 关闭所有敌人；
- 关闭GUI；
- 让玩家无敌；
- 关闭所有游戏逻辑；

46、为每一个足够小的团队，创建一个适合他们的**Debug**选项的**Prefab**。
设置一个用户标识文件，单不要提交到版本库，在游戏运行时读取它。下面是原因：

- 团队的成员不会因为意外的提交了自己的Debug设置而影响到其他人。
- 修改Debug设置不需要修改场景。

47、维护一个包含所有游戏元素的场景。
例如，一个场景，包括所有的敌人，所有可以交互的对象等等。这样可以不用玩很久，而进行全面的功能测试。
48、定义一些**Debug**快捷键常量，并把他们保存在统一的地方。Debug键通常（方便起见）在一个地方来处理，就像其他的游戏输入一样。为了避免快捷键冲突，在一个中心位置定义所有常量。一种替代方案是，在一个地方处理所有按键输入，不管他是否是Debug键。（负面作用是，这个类可能需要引用更多的其他对象）

文档49、为你的设置建立文档。
代码应该拥有最多的文档，但是一些代码之外的东西也必须建立文档。让设计师们通过代码去看如果进行设置是浪费时间。把设置写入文档，可以提高效率（如果文档的版本能够及时更新的话）。
用文档记录下面这些：

- Layer的使用（碰撞、检测、射线检测——本质上说，什么东西应该在哪个Layer里）；
- Tag的使用；
- GUI的depth层级（说什么应该显示在什么之上）；
- 惯用的处理方式；
- Prefab结构；
- 动画Layer。

命名规则和目录结构**50**、遵从一个命名规范和目录结构，并建立文档命名和目录结构的一致性，可以方便查找，并明确指出什么东西在哪里。
你很有可能需要创建自己的命名规则和目录结构，下面的例子仅供参考。

普遍的命名规则

1. 名字应该代表它是什么，例如鸟就应该叫做Bird。
2. 选择可以发音、方便记忆的名字。如果你在制作一个玛雅文化相关的游戏，不要把关卡命名为QuetzalcoatisReturn。
3. 保持唯一性。如果你选择了一个名字，就坚持用它。
4. 使用Pascal风格的大小写，例如**ComplicatedVerySpecificObject**。
   不要使用空格，下划线，或者连字符，除了一个例外（详见为同一事物的不同方面命名一节）。
5. 不要使用版本数字，或者标示他们进度的名词（WIP、final）。
6. 不要使用缩写：DVamp@W应该写成DarkVampire@Walk。
7. 使用设计文档中的术语：如果文档中称呼一个动画为Die，那么使用DarkVampire@Die，而不要用DarkVampire@Death。
8. 保持细节修饰词在左侧：DarkVampire，而不是VampireDark；PauseButton，而不是ButtonPaused。举例说明，在Inspector中查找PauseButton，比所有按钮都以Button开头方便。（很多人倾向于相反的次序，认为那样名字可以自然的分组。然而，名字不是用来分组的，目录才是。名字是用来在同一类对象中可以快速辨识的。）
9. 为一个序列使用同一个名字，并在这些名字中使用数字。例如PathNode0, PathNode1。永远从0开始，而不是1。
10. 对于不是序列的情况，不要使用数字。例如 Bird0, Bird1, Bird2，本应该是Flamingo, Eagle, Swallow。

11. 为临时对象添加双下划线前缀，例如__Player_Backup。

为同一事物的不同方面命名在核心名称后面添加下划线，后面的部分代表哪个方面。例如

- GUI中的按钮状态：EnterButton_Active、EnterButton_Inactive
- 贴图： DarkVampire_Diffuse, DarkVampire_Normalmap
- 天空盒：JungleSky_Top, JungleSky_North
- LOD分组：DarkVampire_LOD0, DarkVampire_LOD1

结构场景组织、工程目录、脚本目录应该使用相似的模式。
目录结构MaterialsGUIEffectsMeshes Actors DarkVampire LightVampire ... Structures Buildings ... Props Plants ... ...PluginsPrefabs Actors Items ...Resources Actors Items ...Scenes GUI Levels TestScenesScriptsTexturesGUIEffects...场景结构[size=11.818181991577148px]CamerasDynamic ObjectsGameplay Actors Items ...GUI HUD PauseMenu ...ManagementLightsWorld Ground Props Structure ...脚本目录结构 ThirdParty ...MyGenericScripts Debug Extensions Framework Graphics IO Math ...MyGameScripts Debug Gameplay Actors Items ... Framework Graphics GUI ...

本主题由 piaolankeke 于 2014-9-5 10:15:11 审核通过

0 0

分享(0) 收藏(1) 举报

发帖 回复 返回列表

高级模式

您需要登录后才可以回帖 登录 | 注册 用QQ帐号登录

回帖后跳转到最后一页