**Unity3D中网格合并示例研究**

Posted on 2013年03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 170 次

[**Unity3D**](http://www.unitymanual.com)中网格合并示例的研究：为了实现游戏人物外形的定制，专门研究了Unity示例程序 。

首先需要了解几个基本对象的结构

一、 SkinedMeshRender：该对象负责网格绘制。

主要数据成员包括：

var bones : Transform[] 骨骼

var materials : Material[] 材质

var sharedMesh : Mesh 网格

其中Mesh的主要成员是：

vertices : Vector3[] 顶点

boneWeights : BoneWeight[] 骨骼权重

boneWeights数组与vertices数组对应，表示对应下标的顶点运动受骨骼影响的权重。BoenWeight结构记录了骨骼在SkinedMeshRender.bones数组中的索引。

二、网格和材质的对应关系

一张实际的网格只能施加一个材质。因此，当render所使用的mesh包含多个实际网格（sub mesh），它对每个sub mesh所施加的材质实际上是materials数组中对应下标的材质。

三、合并网格（CombineMeshes）。函数的第二个参数是设置是否将多个子网格合并成一张实际的网格。正如前面所述，一 个实际的网格只能施加一个材质，所以只有被合并的所有网格原来使用的就是同一个材质（即共享材质）时，将它们真正合并才能正确应用材质。否则，应该将该参 数置为false，表示不实际合并这些sub mesh，而是将它们作为被合并后Mesh对象的sub mesh。

四、数组对应问题：网格顶点和骨骼、sub mesh和材质之间的对应都是通过数组下标进行的，所以操作时保证新生成的个数组下标对应关系正确是非常重要的。

这是例子中组合创建模型的主要函数，我将自己理解后的备注添加在里面。

// Creates a character based on the currentConfiguration recycling a

// character base, this way the position and animation of the character

// are not changed.

// 这个函数实际上并没有将各部分的子网格合并成一张网，而只是将他们合并到

// 同一个Mesh下作为sub mesh。因为一张网格只能用一个材质，只有所有子网格

// 都共享同一个材质时，合并成一张网才能保证材质应用正确。

public GameObject Generate(GameObject root)

{

// The SkinnedMeshRenderers that will make up a character will be

// combined into one SkinnedMeshRenderers using multiple materials.

// This will speed up rendering the resulting character.

List<CombineInstance> combineInstances = new List<CombineInstance>();

List<Material> materials = new List<Material>();

List<Transform> bones = new List<Transform>();

//获得构成骨架的所有Transform

Transform[] transforms = root.GetComponentsInChildren<Transform>();

//一次处理构成身体的各部分

foreach (CharacterElement element in currentConfiguration.Values)

{

//GetSkinnedMeshRenderer()内部Instantiat了一个由该部分肢体Assets构成的

//GameObject，并返回Unity自动为其创建SinkedMeshRender。

SkinnedMeshRenderer smr = element.GetSkinnedMeshRenderer();

//注意smr.materials中包含的材质数量和顺序与下面的sub mesh是对应的

materials.AddRange(smr.materials);

for (int sub = 0; sub < smr.sharedMesh.subMeshCount; sub++)

{

CombineInstance ci = new CombineInstance();

ci.mesh = smr.sharedMesh;

ci.subMeshIndex = sub;

combineInstances.Add(ci);

}

// As the SkinnedMeshRenders are stored in assetbundles that do not

// contain their bones (those are stored in the characterbase assetbundles)

// we need to collect references to the bones we are using

// 网格点与骨骼的对应关系是通过Mesh数据结构中的BoneWeight数组来实现的。该数组

// 与网格顶点数组对应，记录了每个网格点受骨骼（骨骼记录在SinkedMeshRender的bones

// 数组中，按下标索引）影响的权重。

// 而此处，示例程序提供的肢体Assets并不包含骨骼，而是返回骨骼名称。因此，推断

// GetBoneNames()返回的骨骼名称应该与实际骨骼数组的顺序相同。

foreach (string bone in element.GetBoneNames())

{

foreach (Transform transform in transforms)

{

//通过名字找到实际的骨骼

if (transform.name != bone) continue;

bones.Add(transform);

break;

}

}

Object.Destroy(smr.gameObject);

}

// Obtain and configure the SkinnedMeshRenderer attached to

// the character base.

// 至此，combineInstances、bones和materials三个数组中的数据对应关系是正确的。

// 合并时，第二个参数是fals，表示保持子网格不变，只不过将它们统一到一个Mesh里

// 来管理，这样只需采用一个SkinedMeshRender绘制，效率较高。

SkinnedMeshRenderer r = root.GetComponent<SkinnedMeshRenderer>();

r.sharedMesh = new Mesh();

r.sharedMesh.CombineMeshes(combineInstances.ToArray(), false, false);

r.bones = bones.ToArray();

r.materials = materials.ToArray();

return root;

}