**GPU Instancing**

**GPU 实例化**

**Introduction**

**简介**

You can use GPU instancing to draw many identical objects with only a few draw calls.

GPU 实例化仅仅使用很少的draw call去绘制许多相同物体

There are some restrictions which you need to bear in mind:

这是一些需要记住的限制.

Your identical objects need to share the same mesh and the same material. You can, however, add per-instance data; see [Adding per-instance data](http://docs.unity3d.com/540/Documentation/Manual/GPUInstancing.html#AddingPerInstanceData), below, for more information.

相同对象需要共享相同的网格和材质.你可以为每个实例添加数据.参见 [Adding per-instance data](http://docs.unity3d.com/540/Documentation/Manual/GPUInstancing.html#AddingPerInstanceData)详细说明

The MeshRenderer component and Graphics.DrawMesh API are supported.

MeshRenderer组件和Graphics DrawMesh Api是支持的

GPU instancing is available on the following platforms:

GPU 实例化 是可以在以下平台运用的：

Windows: DX11 / DX12 with SM 4.0 and above; OpenGL 4.1 and above

OS X & Linux: OpenGL 4.1 and above

PlayStation 4

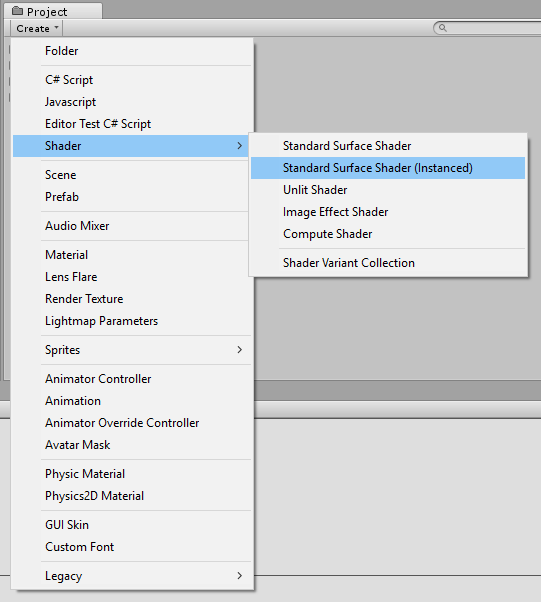
Xbox One

**Adding instancing to your objects**

**添加实例到对象中**

There is a Standard Surface Shader available that supports instancing. Add one to your project by selecting**Shader > Standard Surface Shader (Instanced).**

这是一个支持实例化的标准曲面着色器.添加一个到工程里. 选择**Shader-> Standard Surface Shader(Instanced)**

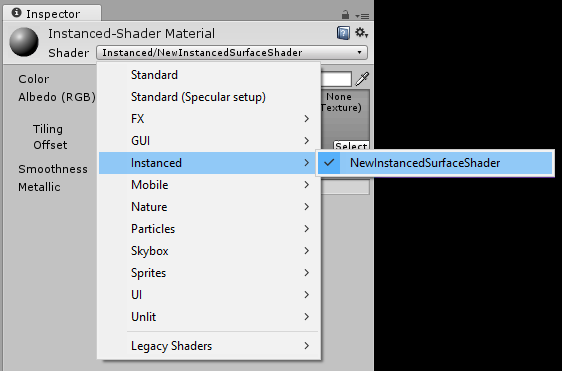


Adding the Standard Instanced Shader

添加标准实例化的着色器

Apply this shader to your GameObject’s Material. In your Material’s Inspector window, click the **Shader** drop-down, roll over the**Instanced**field and choose your instanced shader from the list:

让一个GameObject的材质应用这个着色器.在材质的Inspector窗口上,点击**Shader**,下拉列表选择Instanced字段并且从中选择实例化着色器



Assigning the Standard Instanced Shader to a Material

指定一个材质是标准实例化的着色器.

**Adding per-instance data**

**为每一个实例添加数据**

Even though the instanced objects are sharing the same mesh and material, you can set shader properties on a per-object basis using the **MaterialPropertyBlock** API. In the example below, each object is assigned a random color value using the **\_Color** property:

即使每一个实例化的对象共享相同的Mesh和材质,但你可以在每个对象的基础上使用**MaterialPropertyBlock** API设置材质属性. 在下面这个例子中,每个对象使用**\_Color**属性分配一个随机的颜色.

MaterialPropertyBlock props = new MaterialPropertyBlock();

MeshRenderer renderer;

foreach (GameObject obj in objects)

{ ;

float r = Random.Range(0.0f, 1.0f);

float g = Random.Range(0.0f, 1.0f);

float b = Random.Range(0.0f, 1.0f);

props.SetColor("\_Color", new Color(r, g, b));

renderer = obj.GetComponent<MeshRenderer>();

renderer.SetPropertyBlock(props);

}

**Adding instancing to your own shaders**

**添加实例化到你的着色器中**

Let’s take a simple unlit shader and make it capable of instancing:

让我们看一个简单的不受光的着色器让他能够实例化

Shader "SimplestInstancedShader"

{

Properties

{

\_Color ("Color", Color) = (1, 1, 1, 1)

}

SubShader

{

Tags { "RenderType"="Opaque" }

LOD 100

Pass

{

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#pragma multi\_compile\_instancing

#include "UnityCG.cginc"

struct appdata

{

float4 vertex : POSITION;

UNITY\_INSTANCE\_ID

};

struct v2f

{

float4 vertex : SV\_POSITION;

UNITY\_INSTANCE\_ID

};

UNITY\_INSTANCING\_CBUFFER\_START (MyProperties)

UNITY\_DEFINE\_INSTANCED\_PROP (float4, \_Color)

UNITY\_INSTANCING\_CBUFFER\_END

v2f vert (appdata v)

{

v2f o;

UNITY\_SETUP\_INSTANCE\_ID (v);

UNITY\_TRANSFER\_INSTANCE\_ID (v, o);

o.vertex = UnityObjectToClipPos (v.vertex);

return o;

}

fixed4 frag (v2f i) : SV\_Target

{

UNITY\_SETUP\_INSTANCE\_ID (i);

return UNITY\_ACCESS\_INSTANCED\_PROP (\_Color);

}

ENDCG

}

}

}

**Added code**

**额外的代码**

| **Addition:** | **Function:** |  |
| --- | --- | --- |
| **#pragma multi\_compile\_instancing** | multi\_compile\_instancing generates a shader with two variants: one with built-in keyword **INSTANCING**\_ON defined (allowing instancing), the other with nothing defined. This allows the shader to fall back to a non-instanced version if instancing isn’t supported on the GPU.  multi\_compile\_instancing生成了一个带有两个变体的着色器:一个内置带有定义**INSTANCING\_ON**的关键字 （允许实例化）,另外一个没有定义.  如果GPU不支持实例化将会使着色器退回到不支持的版本 |  |
| **UNITY\_INSTANCE\_ID** | This is used in the vertex shader input/output structure to define an instance ID. See **SV\_InstanceID** for more information.  采用定点着色器的输入输出结构定义实例化ID.参见**SV\_InstanceID**更多详情 |  |
| **UNITY\_INSTANCING\_CBUFFER\_START(name)/ UNITY\_INSTANCING\_CBUFFER\_END** | Every per-instance property must be defined in a specially named constant buffer. Use this pair of macros to wrap the properties you want to be made unique to each instance.  每一个实例化属性必须定义在一个特殊命名的常量缓冲区中.使用这双宏封装每个实例所独特的属性 |  |
| **UNITY\_DEFINE\_INSTANCED\_PROP(float4, color)** | This defines a per-instance shader property with a type and a name. In this example the color property is unique.  每个实例化着色器定义了一个类型和一个名字的属性.在这个例子中颜色是唯一的 |  |
| **UNITY\_SETUP\_INSTANCE\_ID(v);** | This makes the instance ID accessible to shader functions. It must be used at the very beginning of a vertex shader, and is optional for fragment shaders.  使Instance ID可以访问着色器的功能.他必须在顶点或者片元着色器的开端. |  |
| **UNITY\_TRANSFER\_INSTANCE\_ID(v, o);** | This copies the instance ID from the input structure to the output structure in the vertex shader. This is only necessary if you need to access per-instance data in fragment shader.  在顶点着色器中从输入结构复制实例化ID到输出结构.如果你需要在片元着色器中访问每一个实例化数据,这是必要的. |  |
| **UNITY\_ACCESS\_INSTANCED\_PROP(color)** | This accesses a per-instance shader property. It uses instance ID to index into instance data array.  访问每一个实例化属性.使用实例ID索引到实例化数据的数组中. |  |

**Note**: As long as material properties are instanced, renderers can always be rendered instanced, even if you put different instanced properties into different renderers. Normal “non-instanced” properties cannot be batched, so do not put them in the **MaterialPropertyBlock**; instead, create different materials for them.

**注解**：即使你在不同的渲染器中使用不同的实例化属性,只要材质属性是可以被序列化的,渲染器将一直渲染实例化.通常非实例化属性不能分批处理,所以不要把他们放在材质属性块(**MaterialPropertyBlock**)中.相反,为他们创建不同的材质

**A note regarding UnityObjectToClipPos**

**一个关于UnityObjectToClipPos的报告**

**UnityObjectToClipPos(v.vertex)** is always preferred where**mul(UNITY\_MATRIX\_MVP,v.vertex)**would otherwise be used. While you can continue to use **UNITY\_MATRIX\_MVP**as normal in instanced shaders,**UnityObjectToClipPos** is the most efficient way of transforming vertex positions from object space into clip space.

优先使用**UnityObjectToClipPos(v.vertex)**,否则使用**mul(UNITY\_MATRIX\_MVP,v.vertex).**

虽然你可以在实例化的着色器中继续使用**UNITY\_MATRIX\_MVP**,但是**UnityObjectToClipPos**是顶点变换到裁剪空间最有效的方式

In instanced shaders, **UNITY\_MATRIX\_MVP** (among other built-in matrices) is transparently modified to include an extra matrix multiply. **UnityObjectToClipPos** is optimized to perform 2 matrix-vector multiplications simultaneously, and is therefore more efficient than performing the multiplication manually as the shader compiler will not automatically perform this optimization.

在实例化的着色器中,**UNITY\_MATRIX\_MVP**(在其他内置矩阵中)被修改为一个扩展的矩阵乘法

**UnityObjectToClipPos**是优化的二维矩阵乘法.因此他的乘法比手动执行更有效率,编译器不会自动执行此优化

**Further notes**

**进一步说明**

* Instanced draw calls appear in the Frame Debugger as Draw Mesh (instanced).

在Frame Debugger中instanced 作为DrawMesh绘制调用

* When writing or modifying your own shaders, don’t forget to instance shadows, too. For a surface shader, use the **addshadow** option to force the generation of an instanced shadow pass.

当更改你自己的着色器时,也不要忘了实例的阴影.作为一个曲面着色器,使用额外的阴影选项将生成额外的阴影pass

* You don’t have to define per-instance properties, but setting up an instance ID is mandatory, as world matrices need it to work correctly.

你不需要定义每一个实例的属性,但是为了世界矩阵的正常工作,设置instance的ID是必要的.

* When using forward rendering, objects affected by multiple lights can’t be instanced efficiently. Only the base pass can make effective use of instancing, not the add passes.

当使用前向渲染（forward rendering）时,受多个光的影响的物体将不能被有效的实例化.只有基础pass可以有效的实例化,不能是额外的pass

* Objects that use lightmaps, or are affected by different light or reflection probes, can’t be instanced.

物体使用光照贴图、受不同光或者不同的照明将不能被实例化

* If you have more than 2 passes for multi-pass shaders, only the first passes can be instanced. This is because Unity forces the later passes to be rendered together for each object.

如果着色器有两个以上的Pass,只有第一个pass可以被实例化.这是因为Unity强制为每一个物体之后的pass可以一起渲染

You need to tell Unity to always calculate the vertex transformation. To do this, multiply by M first, then by VP (VP \* M \* v) in add passes. This allows you to avoid conflict in the base/first passes caused by small floating point error.

你需要告诉Unity总是计算顶点变换.首先乘M,然后又乘VP(VP \* M \* V)到额外的passes中.这可以避免在基础pass中发生小的浮点数错误.

* Define **UNITY\_USE\_CONCATENATED\_MATRICES** before including UnityCG.cginc. You don’t need to do this for surface shaders, as it is automatically generated.

在包含的UnityCG.cginc之前定义了**UNITY\_USE\_CONCATENATED\_MATRICES**的宏.不需要为表面着色器去写这个矩阵,它将自动生成

* D3D constant buffers have a maximum size of 64KB. For OpenGL, it’s usually 16KB. You will reach this limit if you try to define too many per-instance properties. The shaders may fail to compile or, even worse, the shader compiler might crash. To work around this, you have to balance between the size of the batch and the size of per-instance properties. Defining **UNITY\_MAX\_INSTANCE\_COUNT** with an integer before including any .cginc file allows you to limit the maximum number of instances an instanced draw call can draw. This allows for more properties per instance in the instance constant buffer. You can achieve the same result when using a surface shader with **#pragma instancing\_options maxcount:number.** The default value of this max instance count is 500; for OpenGL, the actual value is one quarter of the value you specify - so 125 by default.

D3D常量缓冲区中最大的大小是64kb.OpenGL通常是16kb.如果你定义了太多的实例化的属性将会达到这个极限.着色器可能会编译失败,更糟糕的是还可能崩溃.必须为每一个属性在大小和数量上平衡.定义的**UNITY\_MAX\_INSTANCE\_COUNT**是一个整形.任何cginc文件允许在限制的最大数量中实例化.在常量缓冲区中允许每一个实例拥有更多的属性.使用一个曲面着色器和**#pragma instancing\_options**将会得到相同的结果. **maxcount:number.**默认最大的instance数量是500.OpenGL中通常是四分之一,也就是125.

* All the shader macros used in the above example are defined in UnityInstancing.cginc. You can find this file in **[Unity folder]\Editor\Data\CGIncludes.**

所有的着色器中使用的宏定义在UnityInstancing.cginc.中,可以在 **[Unity folder]\Editor\Data\CGIncludes**.中找到