GPU instancing

Introduction

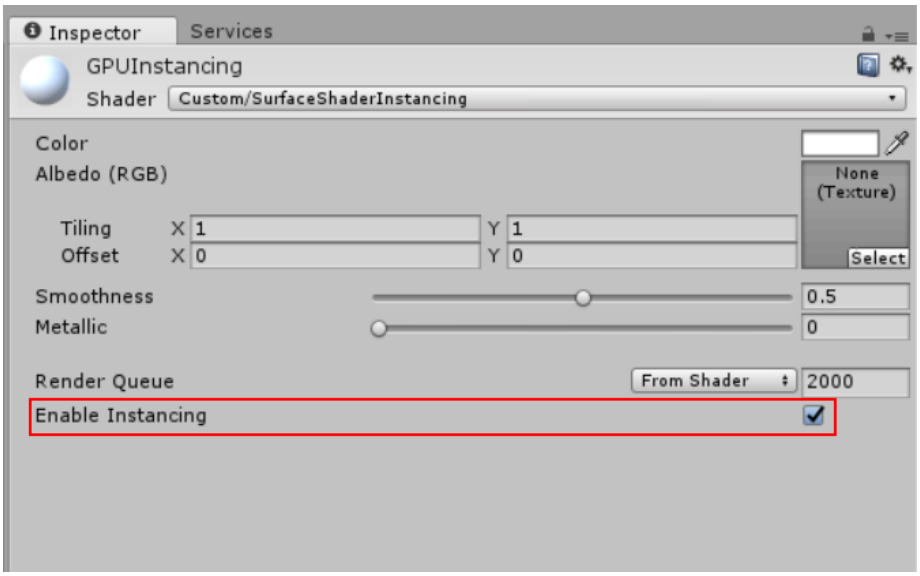
使用GPU Instancing可以一次渲染(render)相同网格的多个副本，仅使用少量DrawCalls。在渲染诸如建筑、树木、草等在场景中重复出现的事物时，GPU Instancing很有用。

每次draw call，GPU Instancing只渲染相同(identical)的网格，但是每个实例(instance)可以有不同的参数(例如，color或scale)，以增加变化(variation)，减少重复的出现。

GPU Instancing可以减少每个场景draw calls次数。这显著提升了渲染性能。

Adding instancing to your Materials

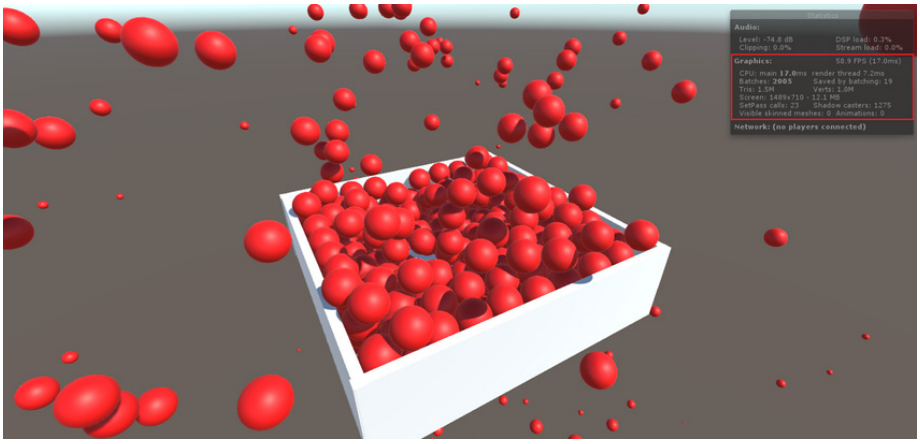
为Materials启用GPU Instancing：在Project window中选中你的Material，然后在**Inspector**窗口中，勾选**Enable Instancing**复选框。



Unity只会在Material使用的Shader支持GPU Instancing时，才会显示这个复选框。这些支持GPU Instancing的Shaders包括：Standard, StandardSpecular 及所有的surface Shaders。更多信息，请查看standard Shaders。

下面的两个屏幕截图展示了具有多个GameObjects的游戏场景。上面的开启了GPU Instancing，下面的没有开启。注意两者中**FPS**，**Batches**和**Saved by batching**的区别。

With GPU Instancing: A simple Scene that includes multiple identical GameObjects that have GPU Instancing enabled


No GPU Instancing: A simple Scene that includes multiple identical GameObjects that do not have GPU Instancing enabled.

当你使用GPU instancing时，受到下面的限制约束：

- Unity自动为instancing调用选择MeshRenderer组件和`Graphics.DrawMesh`。注意不支持SkinnedMeshRenderer。
- 在一次GPU instancing draw call中，Unity仅对共享相同Mesh和相同Material的GameObjects进行batching处理。为了得到更好的instancing效果，尽量使用少量Meshes和Materials。至于如何增加“变化”(variations)，可以在你的shader代码中添加per-instance数据(更多细节，请继续往下看)。

你也可以在c#脚本中调用Graphics.DrawMeshInstanced和Graphics.DrawMeshInstancedIndirect来完成GPU instancing。

GPU Instancing在下面平台和APIs上可用：

- DirectX 11 and DirectX 12 on Windows
- OpenGL Core 4.1+/ES3.0+ on Windows, macOS, Linux, iOS and Android
- Metal on macOS and iOS
- Vulkan on Windows, Linux and Android
- PlayStation 4 and Xbox One
- WebGL (requires WebGL 2.0 API)

Adding per-instance data

默认情况下，Unity在每个instanced draw call中只对具有不同Transforms的GameObjects实例进行batching处理。为了向你的instanced GameObjects添加更多变化(variance)，修改你的Shader，添加per-instance属性(properties)例如材质颜色(material color)。

下面的例子展示了怎样为每一个instance创建一个具有不同颜色值(color values)的instanced Shader。

```
Shader "Custom/InstancedColorSurfaceShader" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
```

```
    }

    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200
        CGPROGRAM
        // Physically based Standard lighting model, and enable shadows on all light types
        #pragma surface surf Standard fullforwardshadows
        // Use Shader model 3.0 target
        #pragma target 3.0
        sampler2D _MainTex;
        struct Input {
            float2 uv_MainTex;
        };
        half _Glossiness;
        half _Metallic;
        UNITY_INSTANCING_BUFFER_START(Props)
           UNITY_DEFINE_INSTANCED_PROP(fixed4, _Color)
        UNITY_INSTANCING_BUFFER_END(Props)
        void surf (Input IN, inout SurfaceOutputStandard o) {
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * UNITY_ACCESS_INSTANCED_PROP(Props, _Color);
            o.Albedo = c.rgb;
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
}

```

当你声明`_Color`为instanced属性(实例属性)时，Unity将从设置在各个GameObjects上的MaterialPropertyBlock类型的对象处获取`_Color`的值，然后把它们(这些GameObjects)放到一个单独的draw call中。对应的C#代码如下:

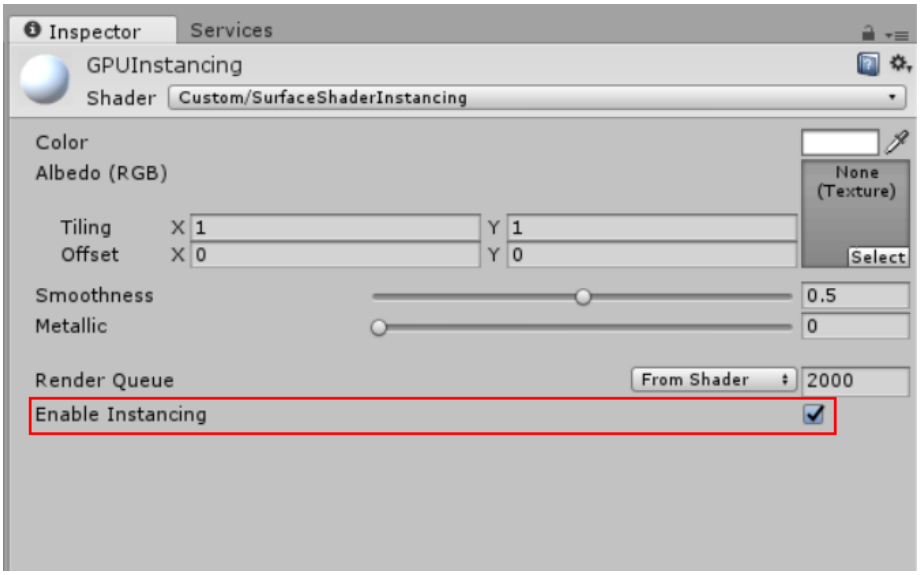
```
MaterialPropertyBlock props = new
MaterialPropertyBlock();
MeshRenderer renderer;

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);
}
```

注意在普通情形下(没有使用instancing sahder或`_Color`不是一个per-instance属性)，draw call batches为被破坏，因为在MaterialPropertyBlock中有不同的值。

为了使这些变化生效，你必须启用GPU Instancing。为了此，中Project window中选中你的材质，然后在Inspector窗口中勾选**Enable Instancing**复选框。



Adding instancing to vertex and fragment Shaders

下面的例子使用了一个简单的unlit Shader 并使它能够用不同的颜色instancing。

```
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_VERTEX_INPUT_INSTANCE_ID
            };

            struct v2f
            {
                float4 vertex : SV_POSITION;
                UNITY_VERTEX_INPUT_INSTANCE_ID // necessary only if you want to access instanced properties in fragment Shader.
            };

            UNITY_INSTANCING_BUFFER_START(Props)
                UNITY_DEFINE_INSTANCED_PROP(float4, _Color)
            UNITY_INSTANCING_BUFFER_END(Props)

            v2f vert(appdata v)
            {
                v2f o;

                UNITY_SETUP_INSTANCE_ID(v);
                UNITY_TRANSFER_INSTANCE_ID(v, o); // necessary only if you want to access instanced properties in the fragment Shader.

                o.vertex = UnityObjectToClipPos(v.vertex);
                return o;
            }

            fixed4 frag(v2f i) : SV_Target
            {
                UNITY_SETUP_INSTANCE_ID(i); // necessary only if any instanced properties are going to be accessed in the fragment Shader.
                return UNITY_ACCESS_INSTANCED_PROP(Props, _Color);
            }
```

```
            ENDCG
        }
    }
}
```

Shader modifications

ADDITION	FUNCTION
#pragma multi_compile_instancing	用来指导Unity生成instancing variants。在surface Shaders中不需要
UNITY_VERTEX_INPUT_INSTANCE_ID	用来在vertex Shader的input/output结构体中定义一个instance ID。更多信息查看`SV_InstanceID`。
UNITY_INSTANCING_BUFFER_START(name) / UNITY_INSTANCING_BUFFER_END(name)	每一个per-instance属性必须定义 在一个特定的命名的constant buffer。用这两对宏来包装那些你想在不同实例中值不同的属性
UNITY_DEFINE_INSTANCED_PROP(float4, `_Color`)	使用给定的type和name来定义一个per-instance Shader属性
UNITY_SETUP_INSTANCE_ID(v);	用来使instance ID在Shader函数中可以被访问。必须用在vertex Shader的最开始处。对于fragment Shaders是可选的
UNITY_TRANSFER_INSTANCE_ID(v, o);	用来在vertex Shader中把instance ID从input structure拷贝到output structure。仅在你需要在fragment Shader中说per-instance数据时使用。

|UNITY_ACCESS_INSTANCED_PROP(arrayName, color)|用来访问一个声明在某个instancing constant buffer中的per-instance Shader属性。它使用一个instance ID来索引到instance data array中。宏中的`arrayName`必须与UNITY_INSTANCING_BUFFER_END(name)中的`name`相匹配。

注意:

- 当用例多个per-instance属性时，你不必把它们都填充到MaterialPropertyBlocks。
- 如果一个实例(instance)缺少某个per-instance属性时，Unity会使用对应的Material中的默认值。如果没有默认值，Unity使0值填充。不要把non-instanced属性放在MaterialPropertyBlock中，因为这会禁用instancing。相反地，为它们创建别的Materials。

Advanced GPU instancing tips

Batching priority

在批处理(batching)时，Unity将静态批处理(Static batching)优先于instancing。如果你把某个GameObject标记为使用static batching，并且Unity成功批处理了它，Uinty会在这个GameObject上禁用instancing，即使它的Renderer使用了instancing Shader。这种情况发生时，Inspector 窗口会显示一条警告，建议你禁用Static Batching。为此，打开**Player**设置(**Edit > Project Settings**)，然后选择**Player**分类，打开**Other Settings**(for your platform)，然后在**Rendering**区域内，禁用**Static Batching**设置。

Unity使instancing的优先级高于**dynamic batching**。如果Unity可以instance一个Mesh，就会禁用在这个Mesh上dynamic batching。

Graphics.DrawMeshInstanced

某些因素可能阻止GameObjects被自动地instanced。这些因素包括材质变化和深度排序。使用Graphics.DrawMeshInstanced可以强制Unity使用GPU instancing来绘制这些物体。和Graphics.DrawMesh相似，这个函数绘制一帧的网格，不需要创建不必要的GameObjects。

Graphics.DrawMeshInstancedIndirect

在脚本中使用`DrawMeshInstancedIndirect`从一个compute buffer中读取instancing draw calls的参数，包括instances的数目。对于填充GPU的所有instance data，这是有用的，而且CPU不知道 要绘制的instances的数目(例如，什么时候完成GPU culling)。更多相关解释和例子，请看Graphics.DrawMeshInstancedIndirect的API 文档。

Global Illumination support

自从Unity 2018.1，GPU Instancing开始使用light probes, occlusion probes(in Shadowmask mode)和lightmap STs来支持全局光照(Global Illumination, GI)渲染。Standard shaders和surface shaders自动开启GI 支持。

受烘焙到场景中的light probes和occlusion probes影响的dynamic renderers和烘焙到相同lightmap贴图的static renderers 可以通过Forward and Deferred 渲染循环(render loop)使用GPU Instancing，自动地被批处理(batched)。

> *原文:Dynamic renderers affected by light probes and occlusion probes baked in the scene, and static renderers baked to the same lightmap texture, can be automatically batched together using GPU Instancing by Forward and Deferred render loop.*

对于Graphics.DrawMeshInstanced，你可以通过设置`LightProbeUsage`参数为`CustomProvided` 来启用light probe和occlusion probe渲染，然后提供一个拷贝了probe data的MaterialPropertyBlock。查看关于LightProbes.CalculateInterpolatedLightAndOcclusionProbes的文档。

Global Illumination and GPU Instancing

Unity中，GPU Instancing支持GI渲染。每个GPU instance可以支持来自不同的Light Probes，某个lightmap(可以用多个atlas regions)，或者某个Light Probe Proxy Volume组件(baked for包含所有instances的空间体(space volume))。Standard shaders和surface shaders默认开启支持。

> *原文: GPU Instancing supports Global Illumination (GI) rendering in Unity. Each GPU instance can support GI coming from either different Light Probes, one lightmap (but multiple atlas regions in that lightmap), or one Light Probe Proxy Volume component (baked for the space volume containing all the instances). Standard shaders and surface shaders come with this support enabled.*

通过Forward和Deferred渲染循环，你可以使用GPU Instancing来自动地批处理(batch) 受到baked Light Probes(包括它们的occlusion data)影响的dynamic Mesh Renderers，或者static Mesh Renderers (baked to the same lightmap Texture)。

对于Graphics.DrawMeshInstanced，你可以通过设置`LightProbeUsage`参数为`CustomProvided`来启用light probe和occlusion probe渲染，然后提供一个拷贝了probe data的MaterialPropertyBlock。查看关于LightProbes.CalculateInterpolatedLightAndOcclusionProbes的文档。

另外一种选择是，你可以向`Graphics.DrawMeshInstanced`传递一个LPPV 组件的引用和 LightProbeUsage.UseProxyVolume 。当你这样做时，所有的instances采样Light Probe数据的L0和L1 bands的volume。

> *原文: Alternatively, you can pass an LPPV component reference and LightProbeUsage.UseProxyVolume to Graphics.DrawMeshInstanced. When you do this, all instances sample the volume for the L0 and L1 bands of the Light Probe data. Use MaterialPropertyBlock if you want to supplement L2 data and occlusion data. For more information, see Light Probes: Technical Information.*

Shader warming-up

从Unity 2017.3开始，如果你想在着色器第一次渲染时获得绝对平滑的渲染效果，你需要预热(warm up)着色器来在OpenGL上使用instancing。若你在一个不要求shader预热的平台上为instancing预热shaders，什么都不会发生。

更多信息查看ShaderVariantCollection.WarmUp和Shader.WarmupAllShaders。

UnityObjectToClipPos

当写shader时，总是用`UnityObjectToClipPos(v.vertex)`而不是`mul(UNITY_MATRIX_MVP,v.vertex)`。

尽管你可以继续在instanced Shaders中正常使用`UNITY_MATRIX_MVP`，然而`UnityObjectToClipPos`是把顶点位置从对象空间(object sapce)转换到裁剪空间(clip space)的最高效的方式。

Further notes

- Surface Shaders默认有instancing varianats产生，除非你在`#pragma surface` directive中指定`noinstancing`。Standard和StandardSpecular Shaders支持instancing，但是除了transforms外没有别的per-instance属性。在surface Shader中，Unity忽略`#pragma multi_compile_instancing`的使用。
- 场景中没有任何GameObject启用GPU Instancing时，Unity剥离(strips)instancing variants。为了覆盖剥离行为，打开Graphics设置(菜单: **Edit > Project Settings**，然后选择**Graphics**分类)，找到**Shader stripping**修改**Instancing Variants**。

（待续...)

参考：

1. GPU instancing