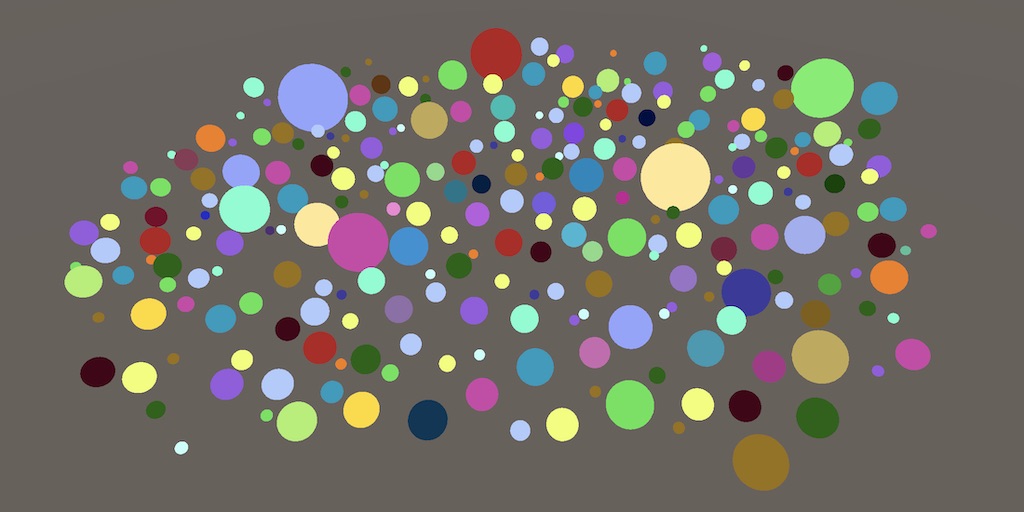
自定义着色器

* 编写HLSL着色器
* 定义常量缓冲区
* 使用unity渲染管线核心库
* 支持动态批处理和GPU实例化

这是Unity可编程渲染管线系列教程的第二章。这章主要介绍了如何用HLSL编写一个着色器，以及怎么在单个draw call中批量高效地渲染多个物体。

本教程使用Unity 2018.3.0f2完成。



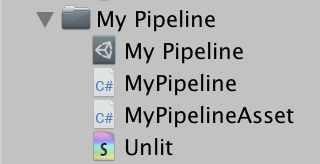
单个draw call绘制256个球形

**一、自定义无光着色器（Unlit Shader）**

虽然我们可以使用默认的unlit shader来测试我们的渲染管线，但是要充分利用自定义渲染管线的强大功能，我们需要为它创建自定义着色器。因此，我们将创建一个自己的着色器来取代Unity的默认unlit shader。

**1.1创建一个着色器**

可以通过“Assets / Create / Shader/Unlit Shader”创建一个着色器。删除创建的文件中所有的默认代码，并命名为Unlit。



Unlit Shader

关于着色器的基础知识在 [Rendering 2, Shader Fundamentals](https://catlikecoding.com/unity/tutorials/rendering/part-2/) 章节中。如果你对此不熟悉，请仔细阅读，以便了解基础知识。一个能够正常运行的着色器至少要定义一个Shader语块，其中必须包含一个Properties语块和一个SubShader语块，而SubShader语块中必须包含一个Pass语块。Shader后面的字符串会展现在材质球的shader下拉菜单中，我们用"My Pipeline/Unlit"来命名它。

Shader "My Pipeline/Unlit" {

Properties {}

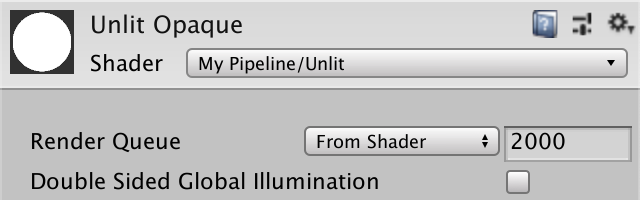
SubShader {

Pass {}

}

}

把Unlit Opaque材质球的shader属性设置成我们刚刚创建的shader。



自定义着色器的Unlit opaque材质球

**1.2 HLSL**

自定义着色器的代码要写在着色器的pass语块中。Unity支持GLSL或HLSL，虽然GLSL用于默认着色器，但是Unity新的渲染管线着色器是使用HLSL编写的，因此我们也是用HLSL。这意味着我们必须将所有代码放在HLSLPROGRAM和ENDHLSL语句之间。

Pass {

HLSLPROGRAM

ENDHLSL

}

Unity着色器至少需要包含一个顶点着色器和一个片元着色器，每个函数都用一个pragma编译器指令定义。我们把定点着色器命名为UnlitPassVertex，片元着色器命名为UnlitPassFragment。但是我们不会直接把这些函数放在着色器文件中。我们把HLSL代码放在一个单独的包含文件中，我们也将其命名为Unlit，但使用hlsl扩展名。把它放在与Unlit.shader相同的文件夹中，然后在pragma指令之后包含HLSL程序。

HLSLPROGRAM

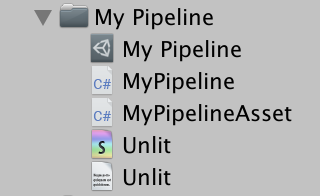
#pragma vertex UnlitPassVertex

#pragma fragment UnlitPassFragment

#include "Unlit.hlsl"

ENDHLSL

不幸的是，Unity没有方便的菜单项来创建HLSL文件。您必须自己创建它，先复制Unlit.shader文件，将文件扩展名更改为hlsl并删除其中的着色器代码。



在文件被包含多次的情况下，为了防止出现重复代码，我们要在包含文件中以包含防护开头。虽然这种情况永远不会发生，但最好始终为每个包含文件执行此操作。

#ifndef MYRP\_UNLIT\_INCLUDED

#define MYRP\_UNLIT\_INCLUDED

#endif // MYRP\_UNLIT\_INCLUDED

至少，在顶点着色器中我们需要知道顶点的坐标，另外它必须输出一个齐次裁剪坐标中的位置。因此，我们在顶点着色器的定义输入和输出结构中都要定义一个float4类型的位置变量。

#ifndef MYRP\_UNLIT\_INCLUDED

#define MYRP\_UNLIT\_INCLUDED

struct VertexInput {

float4 pos : POSITION;

};

struct VertexOutput {

float4 clipPos : SV\_POSITION;

};

#endif // MYRP\_UNLIT\_INCLUDED

接下来，我们定义顶点着色器的函数UnlitPassVertex。目前，我们将直接使用模型空间的顶点位置作为剪辑空间中的顶点位置作为输出。虽然这不正确，但是可以马上得到一个可以正确编译的着色器。我们稍后会添加正确的坐标转换代码。

struct VertexOutput {

float4 clipPos : SV\_POSITION;

};

VertexOutput UnlitPassVertex (VertexInput input) {

VertexOutput output;

output.clipPos = input.pos;

return output;

}

#endif // MYRP\_UNLIT\_INCLUDED

目前我们仍然使用默认的白色作为片元着色器的输出，所以在片元着色器中简单地返回float4类型的1。顶点着色器的输出经过栅化器插值后输入到片元着色器，因此将其作为片元着色器的参数，虽然我们尚未使用它。

VertexOutput UnlitPassVertex (VertexInput input) {

VertexOutput output;

output.clipPos = input.pos;

return output;

}

float4 UnlitPassFragment (VertexOutput input) : SV\_TARGET {

return 1;

}

#endif // MYRP\_UNLIT\_INCLUDED

**1.3转换矩阵**

此时我们拥有了一个能够正确编译的着色器，即使它还不能产生合理的结果。下一步是要把顶点坐标转换为正确的空间。如果我们有一个模型 - 观察 - 投影矩阵，那么我们可以直接从模型空间转换到裁剪空间，但Unity不会为我们创建这样的矩阵。它提供了一个可用的模型矩阵，我们可以使用它从模型空间转换为世界空间。我们需要一个float4x4 unity\_ObjectToWorld变量来存储这个矩阵。当我们使用HLSL时，我们必须自己定义该变量。然后在顶点着色器中把模型顶点坐标转换到时间空间。

float4x4 unity\_ObjectToWorld;

struct VertexInput {

float4 pos : POSITION;

};

struct VertexOutput {

float4 clipPos : SV\_POSITION;

};

VertexOutput UnlitPassVertex (VertexInput input) {

VertexOutput output;

float4 worldPos = mul(unity\_ObjectToWorld, input.pos);

output.clipPos = worldPos;

return output;

}

接下来，我们需要将世界空间转换为裁剪空间。这是通过观察-投影矩阵完成的，Unity是通过float4x4 unity\_MatrixVP变量提供的。添加它然后完成坐标转换。

float4x4 unity\_MatrixVP;

float4x4 unity\_ObjectToWorld;

…

VertexOutput UnlitPassVertex (VertexInput input) {

VertexOutput output;

float4 worldPos = mul(unity\_ObjectToWorld, input.pos);

output.clipPos = mul(unity\_MatrixVP, worldPos);

return output;

}

我们的着色器现在可以正常工作了，所有使用unlit材质球的对象再次可见了，但是全是白色的。由于着色器内部使用转换矩阵和四维的坐标向量相乘，所以我们现在的转换乘法效率不高。坐标向量的第四个分量总是1，通过明确这一点，我们可以使编译器优化计算。

float4 worldPos = mul(unity\_ObjectToWorld, float4(input.pos.xyz, 1.0));

**1.4常量缓冲区**

Unity没有为我们提供模型 - 观察 - 投影矩阵，因为这样可以避免模型矩阵和观察-投影矩阵的乘法。除此之外，观察-投影矩阵可以重复用于在同一帧中使用相同相机绘制的所有内容。因为这一点，Unity的着色器将这些矩阵放在不同的常量缓冲区中。虽然我们将它们定义为变量，但它们的数据在绘制单个物体时保持不变，并且还不知如此。我们把观察-投影矩阵放入per-frame缓冲区，而模型矩阵放入per-draw缓冲区。

虽然并不严格要求将着色器变量放在常量缓冲区中，但这样做可以更有效地更改同一缓冲区中的所有数据。至少，在图形API支持的情况下是这样。但是OpenGL并不支持。

为了尽可能高效，我们要使用常量缓冲区。Unity将VP矩阵放在UnityPerFrame缓冲区中，将模型矩阵放在UnityPerDraw缓冲区中。还有更多的数据放在这些缓冲区中，但我们还不需要它，因此不需要包含。除了cbuffer关键字和变量仍然可以像以前一样访问，常量缓冲区的定义和结构体一样。

cbuffer UnityPerFrame {

float4x4 unity\_MatrixVP;

};

cbuffer UnityPerDraw {

float4x4 unity\_ObjectToWorld;

}

**1.5核心库**

由于常量缓冲区不是在所有平台都能提高效率，因此Unity着色器通过宏控制，在需要的时候打开它们。使用带有name参数的CBUFFER\_START宏而不是直接只用cbuffer关键字，并且在常量缓冲区末尾使用CBUFFER\_END宏。

CBUFFER\_START(UnityPerFrame)

float4x4 unity\_MatrixVP;

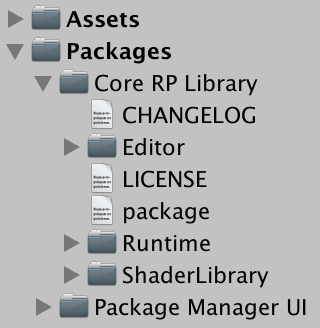
CBUFFER\_END

CBUFFER\_START(UnityPerDraw)

float4x4 unity\_ObjectToWorld;

CBUFFER\_END

由于这两个宏为定义，会导致编译器报错。我们将利用Unity渲染管道核心库来解决这个报错，而不是弄清楚何时应该使用常量缓冲区并自己定义宏。它可以通过包管理器窗口添加到我们的项目中。切换到All Packages列表并在Advanced下启用Show preview packages，然后选择Render-pipelines.core并安装它。我正在使用版本4.6.0预览版，这是Unity 2018.3可使用的最高版本。



现在我们可以通过路径"Packages/com.unity.render-pipelines.core/ShaderLibrary/Common.hlsl"来包含公共库功能。它定义了多个有用的函数和宏，包括常量缓冲区宏，因此在使用它们之前包含它。

#include "Packages/com.unity.render-pipelines.core/ShaderLibrary/Common.hlsl"

CBUFFER\_START(UnityPerFrame)

float4x4 unity\_MatrixVP;

CBUFFER\_END

**1.6编译目标等级**

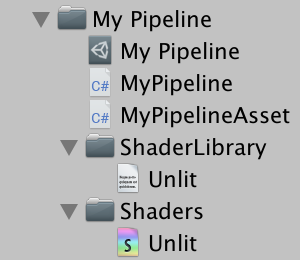
对于大多数平台而言,我们的着色器又能正常工作。在包含公共库之后，我们的着色器在OpenGL ES 2平台编译。这是因为默认情况下，对于OpenGL ES 2平台而言， Unity着色器编译器不支持核心库。我们可以通过添加"#pragma prefer\_hlslcc gles"编译指令到我们的着色器来解决这个问题，这正是Unity的轻量级渲染管线的做法。但是，我们根本不打算支持OpenGL ES 2，因为它仅在老旧的移动设备上才使用。我们通过使用"#pragma target"编译指令把着色器的编译目标等级改成3.5而不是默认的2.5。

#pragma target 3.5

#pragma vertex UnlitPassVertex

#pragma fragment UnlitPassFragment

**1.7文件结构**

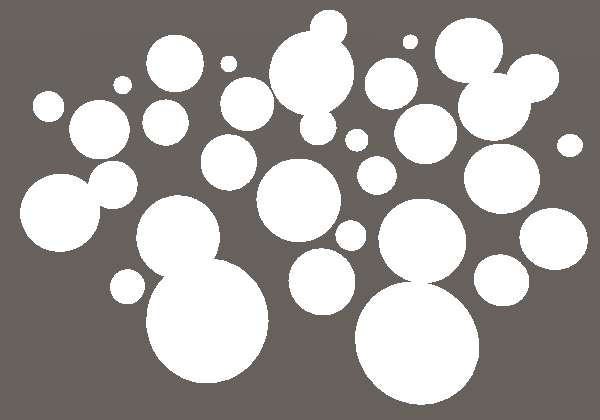
核心库的所有HLSL包含文件都位于ShaderLibrary文件夹中。我们也要这样做，所以在My Pipeline文件夹中创建两个文件夹ShaderLibrary和Shaders，将Unlit.hlsl放在ShaderLibrary文件夹中，将Unlit.shader放在Shaders文件夹中。

同时我们要把色器中包含文件的路径"Unlit.hlsl"改为"../ShaderLibrary/Unlit.hlsl"。

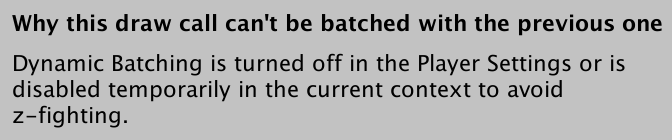
#include "../ShaderLibrary/Unlit.hlsl"

**二、动态批处理**

现在我们有了一个最简单的自定义着色器，我们可以用它来进一步研究我们的管线如何渲染。一个重要的问题是它的效率如何。我们在场景中创建大量的使用我们自定义着色器渲染的球体来测试一下。值得注意的是要保持他们缩放一致。



我们可以通过帧调试器来研究场景是如何被绘制的，你会注意到每个球体都需要单独的绘制调用。这不是很有效，因为每次绘制调用CPU和GPU都需要进行通信，这会导致开销。理想情况下，多个球体可以通过一次调用一起绘制。虽然这是可能的，但目前还没有发生。当您选择其中一个绘制调用时，帧调试器会给我们一个提示。



**2.1启用批处理**

通过帧调试器，我们发现没使用动态批处理，因为它被禁用了或者是深度排序会干扰它。如果您检查播放器设置，那么您将看到确实禁用了动态批处理选项。但是，即使启用它，也没无效。这是因为播放器设置适用于Unity的默认渲染管线，而不是我们的自定义管线。

要为我们的管线启用动态批处理，我们必须在MyPipeline.Render中把draw settings的flags设置为DrawRendererFlags.EnableDynamicBatching。

var drawSettings = new DrawRendererSettings(

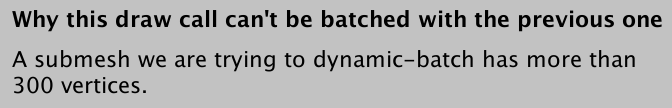
camera, new ShaderPassName("SRPDefaultUnlit")

);

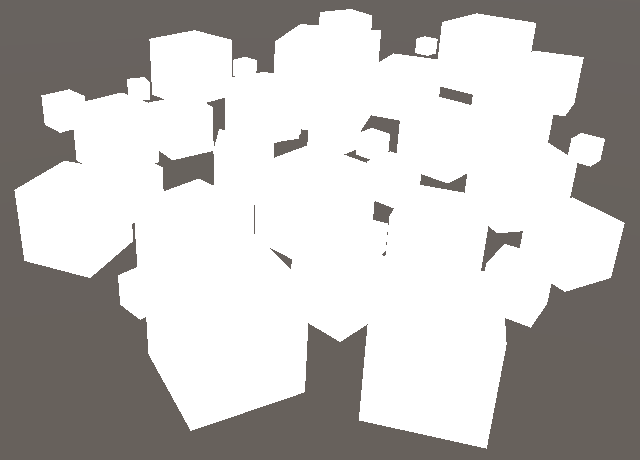
drawSettings.flags = DrawRendererFlags.EnableDynamicBatching;

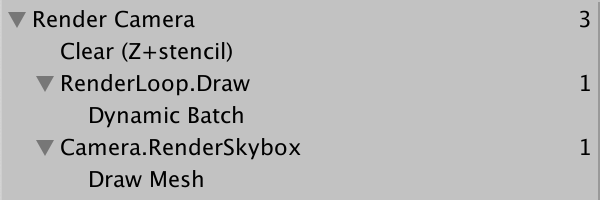
drawSettings.sorting.flags = SortFlags.CommonOpaque;

修改之后，我们发现仍然没有使用动态批处理，但是原因以及变了。动态批处理意味着Unity在绘制之前将物体合并在一个mesh中。这需要消耗每帧的CPU时间并且物体的顶点数不能超过300。



球体顶点数太多，立方体顶点数少可以使用动态批处理。因此，把所有物体改成立方体网格。你可以全选，并一次调整它们的网格过滤器。





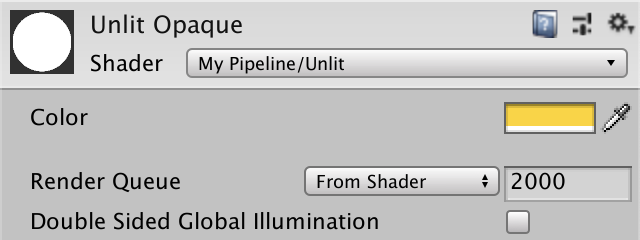
**2.2颜色**

动态批处理适用于使用相同材质绘制的小网格（顶点较少的物体）。但是当涉及多种材质球时，事情变得更加复杂。为了说明这一点，我们需要更改无光着色器的颜色。将color属性添加到它的Properties语块中，并命名为\_Color，"Color"作为标签， 白色作为默认值。

Properties {

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

}



现在我们可以调整材质的颜色了，但它还不会影响绘制。将一个float4 \_Color变量添加到Unlit.hlsl文件中并在片元着色器中返回该变量而不是固定值。颜色是根据材质球定义的，因此可以放在一个常量缓冲区中，只需要在切换材质球时更改。我们将缓冲区命名为UnityPerMaterial。

CBUFFER\_START(UnityPerDraw)

float4x4 unity\_ObjectToWorld;

CBUFFER\_END

CBUFFER\_START(UnityPerMaterial)

float4 \_Color;

CBUFFER\_END

struct VertexInput {

float4 pos : POSITION;

};

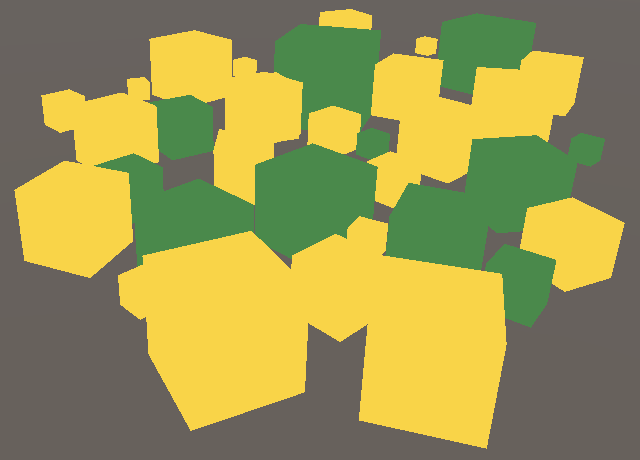
…

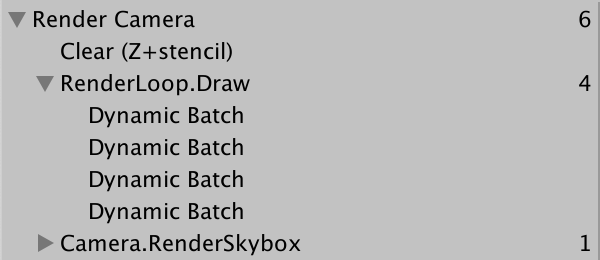
float4 UnlitPassFragment (VertexOutput input) : SV\_TARGET {

return \_Color;

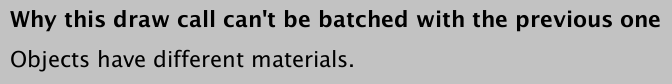
}

复制材质球并设置成不同的颜色，以便我们区分它们。然后选择一些对象并让它们使用新材质球。





动态批处理的批次变多了。因为不同的材质球需要不同的per-material数据，所以每种材质球至少需要一个批次。但是通常会有更多的批次，因为Unity会在空间上对物体进行分组以减少重复绘制。

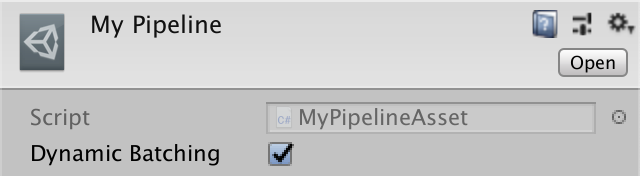


**2.3批处理选项**

动态批处理可能产生好处，但也可能最终并没有什么用，甚至减慢速度。如果场景中不包含许多共享相同材质球的小网格（顶点较少的物体），则禁用动态批处理可能是有意义的，因此Unity就不必每帧判断是否使用动态批处理。因此，我们将添加一个选项，以便为我们的管线启用或禁用动态批处理。我们不能依赖播放器设置，而是需要添加了一个配置字段到我们的MyPipelineAsset，因此我们可以通过编辑器中的管线资源文件对其进行配置。

[ SerializeField ]

bool dynamicBatching;



当MyPipeline实例被创建时，我们要告诉它是否使用动态批处理。我们将在调用其构造函数时将此信息作为参数传过去。

protected override IRenderPipeline InternalCreatePipeline () {

return new MyPipeline(dynamicBatching);

}

因此，我们不能再依赖于MyPipeline的默认构造函数了。需要为MyPipeline添加一个公共构造函数方法，使用布尔参数来传递动态批处理设置，然后添加一个DrawRendererFlags类型的变量drawFlag。在构造函数设置为drawFlag赋值。

DrawRendererFlags drawFlags;

public MyPipeline (bool dynamicBatching) {

if (dynamicBatching) {

drawFlags = DrawRendererFlags.EnableDynamicBatching;

}

}

将drawFlag复制给Render函数中的drawSettings.flags。

drawSettings.flags = drawFlags;

当我们在编辑器中切换管线资源的动态批处理选项时，Unity的批处理行为会立即发生变化。每次我们调整管线资源时，都会创建一个新的管线实例。

**三、GPU实例化**

动态批处理不是我们可以减少每帧绘制调用次数的唯一方法。另一种方法是使用GPU实例化，在这种情况下，CPU通过单个调用告诉GPU需要多次绘制指定的网格和材质的组合。这使得可以对使用相同网格和材质的对象进行分组，而无需构造新网格。这也消除了网格大小的限制。

**3.1GPU实例化选项**

默认情况下GPU实例化是开启的，但我们要自定义一个标记来控制它。让GPU实例化成为可选项，这样可以很容易地比较GPU实例化在启用和禁用下的不同结果。在MyPipelineAsset中添加一个可配置字段，并将其传递给MyPipeline的构造函数。

[SerializeField]

bool instancing;

protected override IRenderPipeline InternalCreatePipeline () {

return new MyPipeline(dynamicBatching, instancing);

}

在MyPipeline的构造函数中，我们需要设置GPU实例化标志。在这种情况下，标志值是DrawRendererFlags.EnableInstancing，把该值和之前的drawFlags做逻辑或运算，如此可以同时启用动态批处理和GPU实例化。当它们都被启用时，Unity倾向于使用GPU实例化。

public MyPipeline (bool dynamicBatching, bool instancing) {

if (dynamicBatching) {

drawFlags = DrawRendererFlags.EnableDynamicBatching;

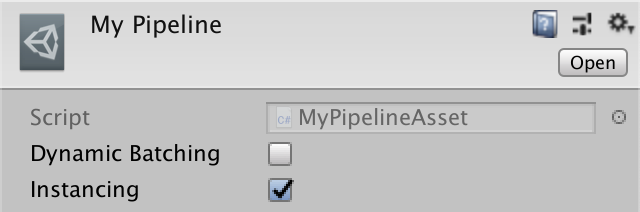
}

if (instancing) {

drawFlags |= DrawRendererFlags.EnableInstancing;

}

}



**3.2材质球支持**

当我们的渲染管线启用GPU实例化时，并不意味着渲染的物体就能实现GPU实例化，它还要求渲染的物体所使用的材料支持GPU实例化。因为并不总是需要GPU实例化，所以它是一个可选项，这需要两个着色器变量：一个支持实例化，另一个不支持实例化。我们可以把#pragma multi\_compile\_instancing指令添加到着色器来创建所必需的变量。在我们的例子中，会产生两个着色器变量，一个启用INSTANCING\_ON宏，另一个禁用INSTANCING\_ON宏。

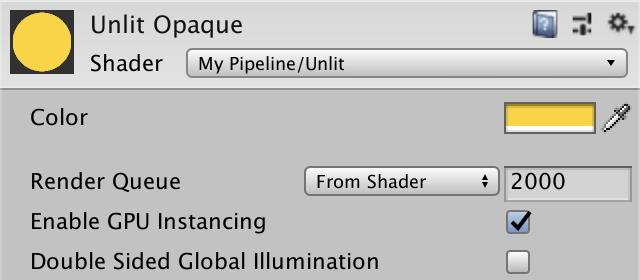
#pragma target 3.5

#pragma multi\_compile\_instancing

#pragma vertex UnlitPassVertex

#pragma fragment UnlitPassFragment

添加新的指令之后，我们的材质球中会出现一个新的可配置选项：启用GPU实例化。



**3.3着色器支持**

启用GPU实例化时，会告诉GPU使用相同的常量数据多次绘制相同的网格。但模型矩阵是该数据的一部分。这意味着我们最终还是不能在一个绘制调用中多次渲染相同的网格。要解决该问题，必须将包含所有对象的模型矩阵的数组放入常量缓冲区中。每个实例都使用自己的索引绘制，通过该索引可以从数组中获取正确的模型矩阵。

在禁用实例化是需要使用unity\_ObjectToWorld矩阵，在启用实例化时需要使用矩阵数组。为了使顶点着色器在两种情况下的的代码保持一致，我们将为模型矩阵定义一个宏：UNITY\_MATRIX\_M。我们之所以这样命名，是因为核心库的包含文件中已经定义了该宏来支持GPU实例化。

CBUFFER\_START(UnityPerDraw)

float4x4 unity\_ObjectToWorld;

CBUFFER\_END

#define UNITY\_MATRIX\_M unity\_ObjectToWorld

…

VertexOutput UnlitPassVertex (VertexInput input) {

VertexOutput output;

float4 worldPos = mul(UNITY\_MATRIX\_M, float4(input.pos.xyz, 1.0));

output.clipPos = mul(unity\_MatrixVP, worldPos);

return output;

}

在我们把模型矩阵定义为UNITY\_MATRIX\_M之后，添加包含文件：UnityInstancing.hlsl，因为它可以在需要的情况下重新定义宏来支持GPU实例化。

#define UNITY\_MATRIX\_M unity\_ObjectToWorld

#include "Packages/com.unity.render-pipelines.core/ShaderLibrary/UnityInstancing.hlsl"

当使用实例化时，当前正在绘制的对象的索引会被GPU添加到其顶点数据。UNITY\_MATRIX\_M依赖于索引，所以我们必须把它添加到VertexInput结构中。我们可以使用UNITY\_VERTEX\_INPUT\_INSTANCE\_ID宏来帮助我们获取索引。

struct VertexInput {

float4 pos : POSITION;

UNITY\_VERTEX\_INPUT\_INSTANCE\_ID

};

最后，在使用UNITY\_MATRIX\_M 宏之前，我们必须通过使用UNITY\_SETUP\_INSTANCE\_ID宏（该宏该收input作为参数）来设置索引ID。

VertexOutput UnlitPassVertex (VertexInput input) {

VertexOutput output;

UNITY\_SETUP\_INSTANCE\_ID(input);

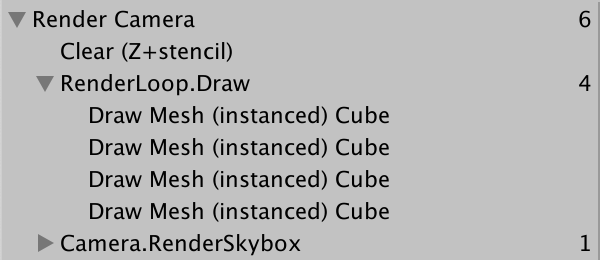
float4 worldPos = mul(UNITY\_MATRIX\_M, float4(input.pos.xyz, 1.0));

output.clipPos = mul(unity\_MatrixVP, worldPos);

return output;

}

现在立方体实现来GPU实例化。像动态批处理一样，我们最终还是有多个批次，那是因为我们使用了不同的材质球。需要确保使用的所有材质球都启用了GPU实例化。



除了模型-世界的矩阵之外，世界-模型矩阵也被默认添加在实例化缓冲区中。这些是模型矩阵的逆矩阵，在我们只使用统一缩放的情况下，我们不需要这些额外的矩阵。我们可以通过把#pragma instancing\_options assumeuniformscaling指令添加到我们的着色器来从缓冲区中去掉世界-模型矩阵。

#pragma multi\_compile\_instancing

#pragma instancing\_options assumeuniformscaling

如果确实支持非统一缩放，则必须使用未启用此选项的着色器。

**3.4不同颜色**

如果我们想在场景中使用多种颜色，我们需要制作不同的材质球，这意味着批次会增加。既然矩阵可以放在数组中，那么颜色应该也可以。所以可以在一个批次中绘制不同颜色的对象。

第一步，我们需要为每个物体单独设置颜色。我们不能通过材质球来做到这一点，因为这是所有物体共享的资源。我们需要创建一个组件，命名为InstancedColor，给它一个可配置的颜色字段。由于它不是自定义渲染管线的一部分，所以把该文件放在My Pipeline文件夹之外。

using UnityEngine;

public class InstancedColor : MonoBehaviour {

[SerializeField]

Color color = Color.white;

}

为了覆盖材质球的颜色，我们需要创建一个MaterialPropertyBlock对象，通过SetColor方法设置它的"\_Color "属性，然后通过调用MeshRenderer的SetPropertyBlock方法将它传递给物体的材质球。我们假设颜色在播放模式中不会改变，所以在Awake方法中进行这些操作。

void Awake () {

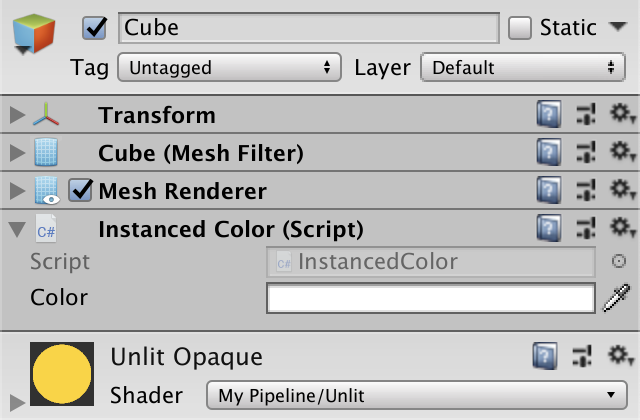
var propertyBlock = new MaterialPropertyBlock();

propertyBlock.SetColor("\_Color", color);

GetComponent<MeshRenderer>().SetPropertyBlock(propertyBlock);

}

把我们的组件添加到场景中的一个对象上。在进入播放模式后你能看到它的颜色发生变化。



要在编辑模式下立即查看场景中的颜色更改，需要把设置颜色的代码移动到OnValidate方法中，然后在Awake中调用OnValidate方法。

void Awake () {

OnValidate();

}

void OnValidate () {

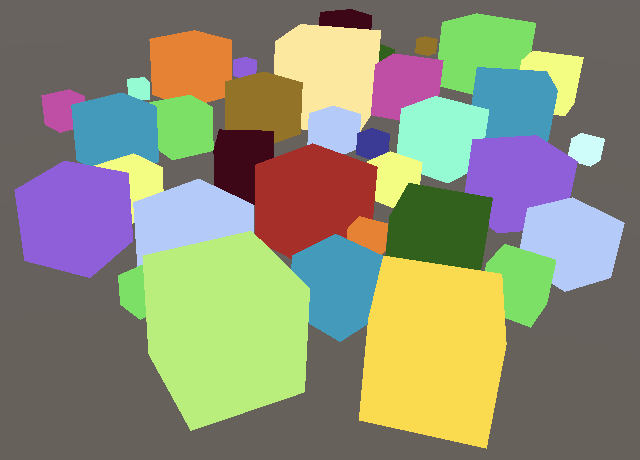
var propertyBlock = new MaterialPropertyBlock();

propertyBlock.SetColor("\_Color", color);

GetComponent<MeshRenderer>().SetPropertyBlock(propertyBlock);

}

把组件添加到所有物体上，同时确保他们使用相同的材质球，然后给不同的物体设置不同的颜色，注意不能在物体上重复添加该组件。



每次通过MaterialPropertyBlock设置颜色时，我们都会创建一个新实例。这不是必需的，因为每个MeshRenderer在内部会生成一个属性块的拷贝来追踪被覆盖的属性。这意味着我们可以重复使用它，因此生成一个静态的属性块，仅在需要时创建它。

static MaterialPropertyBlock propertyBlock;

…

void OnValidate () {

if (propertyBlock == null) {

propertyBlock = new MaterialPropertyBlock();

}

propertyBlock.SetColor("\_Color", color);

GetComponent<MeshRenderer>().SetPropertyBlock(propertyBlock);

}

此外，我们可以通过Shader.PropertyToID方法来获取属性ID，从而略微加快color属性的匹配。每个着色器属性名称都会生成一个全局整数标识。这些标识在单个会话期间始终保持不变，即编译和播放之间。所以我们获取一次，把它赋值给一个静态字段。

static int colorID = Shader.PropertyToID("\_Color");

…

void OnValidate () {

if (propertyBlock == null) {

propertyBlock = new MaterialPropertyBlock();

}

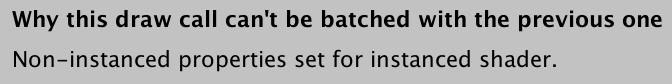
propertyBlock.SetColor(colorID, color);

GetComponent<MeshRenderer>().SetPropertyBlock(propertyBlock);

}

**3.5 Per-Instance不同颜色**

为每个物体设置不同的颜色会影响GPU实例化。虽然我们使用的是相同的材质球，但是用于渲染的数据（颜色）却不同。当使用不同颜色时，会导致每个物体被单独绘制。



为了是GPU实例化再次起作用，需要把颜色数据放在一个数组中。\_Color属性需要和模型矩阵一样处理。在这种情况下，我们需要自己处理，因为核心库不会为自定义属性重新定义宏。我们通过UNITY\_INSTANCING\_BUFFER\_START宏和对应的结束宏手动创建一个用于GPU实例化的常量缓冲区，并且命名为PerInstance。在缓冲区内，我们将颜色定义为**UNITY\_DEFINE\_INSTANCED\_PROP**(**float4**, \_Color)，当禁用GPU实例化时该宏就相当于float4 \_Color，当启用GPU实例化时我们会得到一组实例数据。

//CBUFFER\_START(UnityPerMaterial)

//float4 \_Color;

//CBUFFER\_END

UNITY\_INSTANCING\_BUFFER\_START(PerInstance)

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

UNITY\_INSTANCING\_BUFFER\_END(PerInstance)

为了访问颜色值的代码在启用或者禁用GPU实例化时能保持一致，我们通过UNITY\_ACCESS\_INSTANCED\_PROP宏来获取颜色值，把缓冲区和属性的名称传给该宏。

float4 UnlitPassFragment (VertexOutput input) : SV\_TARGET {

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

}

现在我们必须在片元着色器中使用实例索引。把UNITY\_VERTEX\_INPUT\_INSTANCE\_ID添加到到VertexOutput，然后使用UNITY\_SETUP\_INSTANCE\_ID宏来设置索引ID。最后，我们使用UNITY\_TRANSFER\_INSTANCE\_ID宏把索引从顶点输入复制到顶点输出。

struct VertexInput {

float4 pos : POSITION;

UNITY\_VERTEX\_INPUT\_INSTANCE\_ID

};

struct VertexOutput {

float4 clipPos : SV\_POSITION;

UNITY\_VERTEX\_INPUT\_INSTANCE\_ID

};

VertexOutput UnlitPassVertex (VertexInput input) {

VertexOutput output;

UNITY\_SETUP\_INSTANCE\_ID(input);

UNITY\_TRANSFER\_INSTANCE\_ID(input, output);

float4 worldPos = mul(UNITY\_MATRIX\_M, float4(input.pos.xyz, 1.0));

output.clipPos = mul(unity\_MatrixVP, worldPos);

return output;

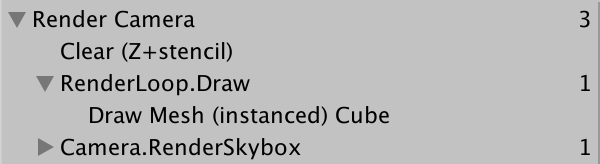
}

float4 UnlitPassFragment (VertexOutput input) : SV\_TARGET {

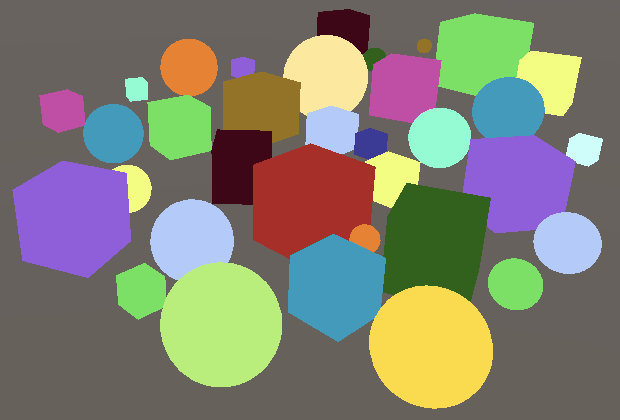
UNITY\_SETUP\_INSTANCE\_ID(input);

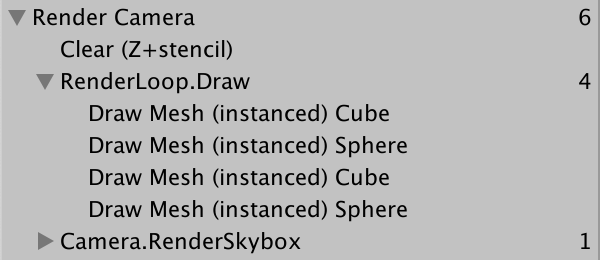
return UNITY\_ACCESS\_INSTANCED\_PROP(PerInstance, \_Color);

}



所有对象最终组合在一个绘制调用中，即使它们都使用不同的颜色。但是，在常量缓冲区中可以放入多少数据是有限制的。GPU实例化批处理的限制取决于每个实例的数据变化量，除此之外，缓冲区的最大值因平台而异。当然仍然只能在使用相同的网格和材质球的情况下才是实现GPU实例化批处理。例如，当同时使用立方体和球体网格时，会分批次。





至此，我们有一个最简单的着色器，并且能够尽可能高效地绘制多个对象。接下来，我们将在此基础上构建更高级的着色器。

下一章我们实现光照。

本章教程项目仓库。