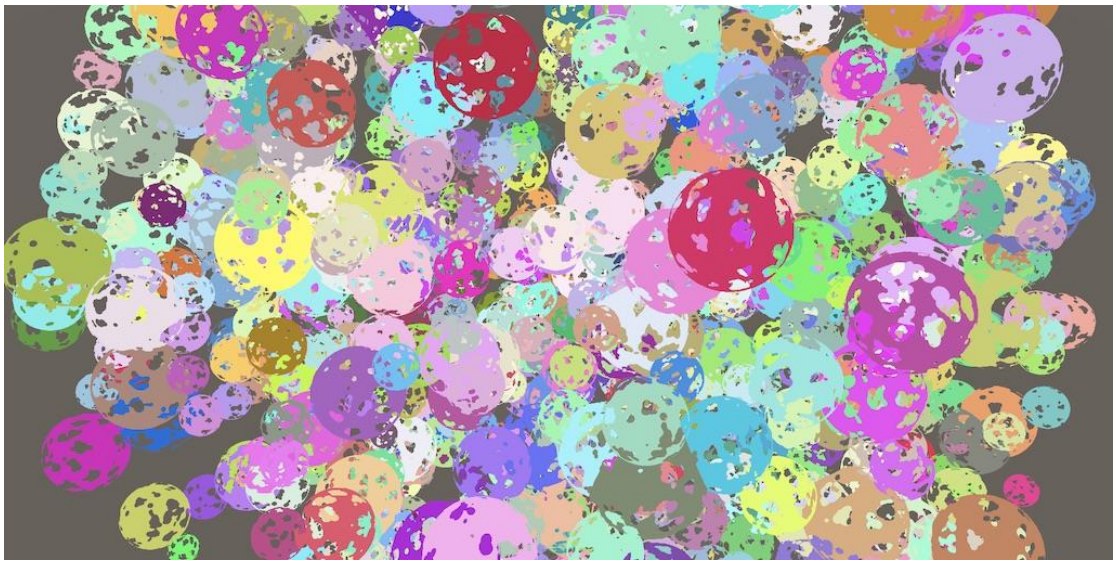
绘制指令

着色器和批次

编写一个 HLSL 着色器
支持 SRP batcher，GPU 实例化和动态合批
为单个物体配置材质属性，并随机绘制多份
创建透明材质和镂空材质

这是创建自定义渲染管线系列教程的第二部分，包含着色器的编写和多个对象的高效绘制。

该教程基于 Unity 2019.2.9f1。



很多球体，但只用了少量的绘制

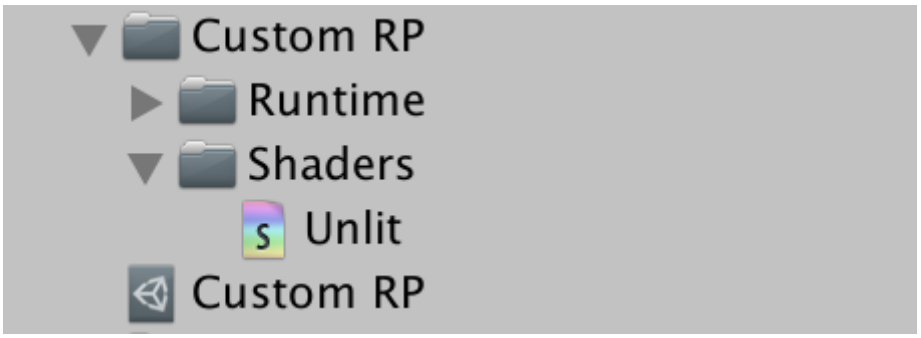
1. 着色器

为了绘制物体，CPU 需要告诉 GPU 需要绘制什么，并且怎样绘制。绘制的物体通常是一个网格。怎样绘制由一套 GPU 指令来决定，我们称之为着色器。除了网格意外，着色器还需要额外的一些信息来完成工作，包括物体的变换矩阵和材质属性。

Unity 的轻量级(LWRP)/通用渲染管线(URP)和高精度渲染管线(HDRP)支持使用 Shader Graph 包来设计着色器，它可以帮助你生成着色器代码。但是我们的自定义渲染管线不支持，所以不得不自己写着色器代码。这有助于我们完全控制着色器，并了解它具体的作用。

1.1. 无光照着色器

我们的第一个着色器将简单的绘制一个不受光照的纯色网格。可以通过 Assets/Create/Shader 菜单来创建一个着色器资源。Unlit Shader 是最合适的，但我们准备从头开始，所以从创建的着色器文件里把所有的默认代码删掉。把文件命名为 Unlit，然后放到 Custom RP 下一个新建的 Shaders 文件夹里。



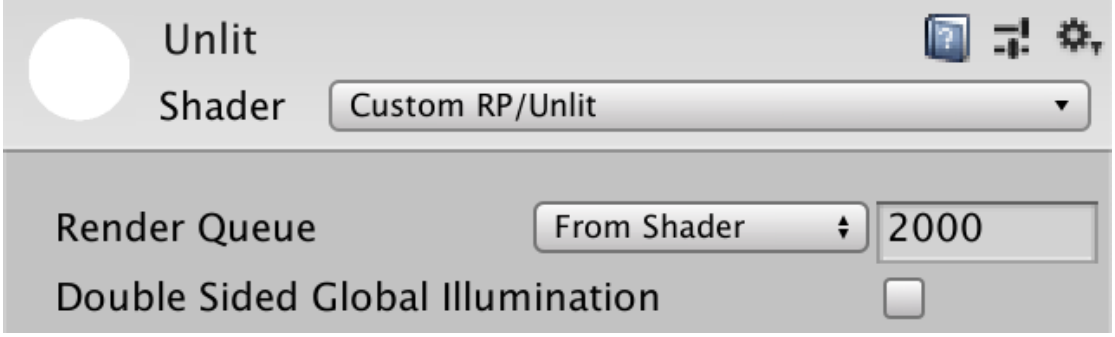
Unlit shader

着色器代码大部分看起来和 C#代码很像，但它包含一些不同的方法，包括一些以前有用但现在无效的老旧方法。

着色器定义与 class 类似，但后面带有一个着色器关键字的字符串，用于在材质的着色器下拉菜单中创建一个入口。我们使用 Custom RP/Unlit。紧接着是一个代码块，包含更多前面带有关键字的代码块。Properties 块定义了材质属性，后面的 SubShader 块里面必须包含一个 Pass 块，用来定义渲染方法。创建这样的结构，其他部分用空结构块。

```
1.  Shader "Custom RP/Unlit" {
2.
3.      Properties {}
4.
5.      SubShader {
6.
7.          Pass {}
8.      }
9.  }
```

这样就定义了一个可编译的最小着色器，并且可以创建一个使用它的材质。



自定义 Unlit 材质

默认的着色器实现将网格渲染成纯白色。材质显示了渲染队列的默认属性，该属性自动从着色器中获取并设置为 2000，这是不透明几何体的默认属性。另外还有一个控制双面整体照明的开关，但与我们无关。

1.2. HLSL 程序

我们编写着色器使用的语言是高级着色器语言(High-Level Shading Language)，简称 HLSL。我们需要将其放在 Pass 块中，在 HLSLPROGRAM 和 ENDHLSL 关键字之间。这么做是必须的因为 Pass 块中也可能放入非 HLSL 语言的代码。

```
1. Pass {
2.     HLSLPROGRAM
3.     ENDHLSL
4. }
```

为了绘制网格，GPU 必须将所有的三角面栅格化，转换成像素数据。它通过将 3D 空间中的顶点坐标转换到 2D 可视空间，再对所得三角形所覆盖的所有像素进行填充来实现。这两步由独立的着色器程序控制，我们都需要去定义它们。第一个被称作顶点内核/程序/着色器，第二个叫做片元内核/程序/着色器。一个片元对应一个显示像素或纹理纹素，尽管可能不会表现出最终的结果，因为它可能会被之后绘制在其上的内容覆盖。

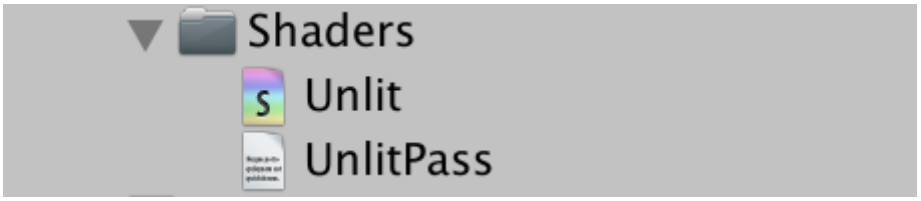
我们使用 pragma 指令为两个程序命名。这些是单行声明，由#pragma 开始，然后是 vertex 或 fragment，再加上相对应的名字。我们使用 UnlitPassVertex 和 UnlitPassFragment。

```
1. HLSLPROGRAM
2. #pragma vertex UnlitPassVertex
3. #pragma fragment UnlitPassFragment
4. ENDHLSL
```

着色器编译器现在显示无法找到声明的着色器内核。我们必须编写同名 HLSL 方法来定义他们的实现。可以在 pragma 指令下面直接写，但我们准备把所有 HLSL 代码放到一个单独的文件里。这次我们使用同文件夹里的 UnlitPass.hlsl 文件。我们可以通过添加#include 指令加上一个文件的相对路径来指示着色器编译器插入该文件的内容。

```
1. HLSLPROGRAM
2. #pragma vertex UnlitPassVertex
3. #pragma fragment UnlitPassFragment
4. #include "UnlitPass.hlsl"
5. ENDHLSL
```

Unity 不能通过方便的菜单操作来创建 HLSL 文件，所以你必须进行复制着色器文件那样类似的操作，将其重命名为 UnlitPass，把文件后缀名改成 hlsl，并清除其中内容。



UnlitPass HLSL 资源文件

1.3. 引用保护

HLSL 文件像 C#类一样用于对代码进行分组，尽管 HLSL 没有类的概念。除了代码块的局部作用域以外，只有一个全局作用域，所以所有内容在任何地方都是可用的，包含文件也与使用命名空间不同。它在 include 指令的位置插入整个文件内容，所以如果你引用了相同文件多次，就会得到重复的代码，这通常会导致编译错误。为了避免这种情况，我们需要在 UnlitPass.hlsl 里添加一个引用保护。

可以使用#define 指令来定义任何标识符，标识符通常定为大写。我们在文件开始定义 CUSTOM_UNLIT_PASS_INCLUDED。

```
1. #define CUSTOM_UNLIT_PASS_INCLUDED
```

这是一个定义了标识符的简单宏的范例。如果它存在，那么意味着文件已经被引入，所以我们不再需要引入。换句话说，我们只想在它尚未被定义时插入代码。我们可以用#ifndef 在定义宏之前进行检查。

```
1. #ifndef CUSTOM_UNLIT_PASS_INCLUDED
2. #define CUSTOM_UNLIT_PASS_INCLUDED
```

如果宏已经被定义，那么在#ifndef 之后的所有代码都会跳过，并且不会被编译。我们必须通过在文件尾添加#endif 指令来结束宏作用域。

```
1. #ifndef CUSTOM_UNLIT_PASS_INCLUDED
2. #define CUSTOM_UNLIT_PASS_INCLUDED
3. #endif
```

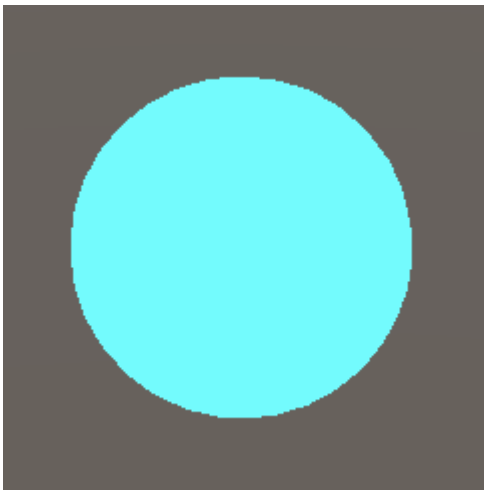
现在我们可以保证文件中所有相关的代码都不会被多次插入，即使我们进行了多次引入。

1.4. 着色器方法

我们在引用保护的作用域内定义着色器方法。写法与 C#方法类似，只是没有访问标识符。从简单的没有任何操作的空方法开始。

```
#ifndef CUSTOM_UNLIT_PASS_INCLUDED
#define CUSTOM_UNLIT_PASS_INCLUDED

void UnlitPassVertex () {}

void UnlitPassFragment () {}

#endif
```

这样就可以对着色器进行编译了。编译结果是一个默认的青色着色器。



青色球体

我们可以通过让片元方法返回一个不同的值来改变颜色。颜色由红、绿、蓝和透明度四个分量所组成的四分量 float4 向量所表示。我们可以用 float4(0.0, 0.0, 0.0, 0.0)来定义黑色，同样可以只写一个 0，因为单个值可以自动扩展到整个向量。透明度现在没有影响，因为我们创建的是一个不透明着色器，所以零也可以。

```
float4 UnlitPassFragment () {
	return 0.0;
}
```

这时着色器编译会报错，因为我们的方法缺少语义。我们必须表明返回的值是什么意义，因为我们可能会产生很多不同含义的数据。目前我们提供的是渲染目标的默认系统值，通过在 UnlitPassFragment 参数之后添加一个冒号和 SV_TARGET 来表明。

```
float4 UnlitPassFragment () : SV_TARGET {
	return 0.0;
}
```

UnlitPassVertex 负责转换顶点坐标，所以应该返回一个坐标。坐标同样是一个 float4 向量因为它被定义为一个齐次裁剪空间坐标，但我们稍后再处理。同样地，我们先使用零向量，并且用 SV_POSITION 来表明语义。

```
float4 UnlitPassVertex () : SV_POSITION {
	return 0.0;
}
```

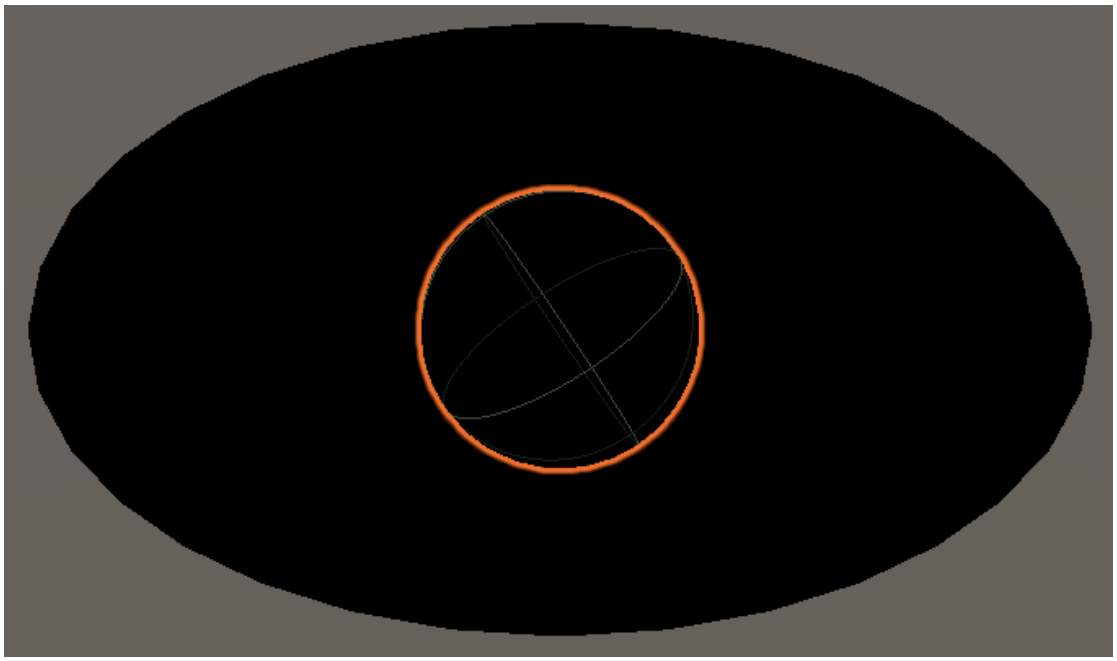
1.5. 空间变换

当所有顶点都被设置成 0 时，网格就坍缩成一个点并且不会被渲染。顶点方法的主要作用就是将原始的顶点坐标转换到正确的空间中。如果有需求，该方法在调用时可以获取可用的顶点数据，我们通过给 UnlitPassVertex 添加参数来实现。我们需要在物体空间中定义的顶点坐标，所以我们将它命名为 positionOS，与 Unity 的新渲染管线相同。坐标的类型是 float3，因为是一个 3D 点。让我们直接把这个值加上 1 作为第四个分量组成 float4(positions, 1.0)作为返回值。

```
float4 UnlitPassVertex (float3 positionOS) : SV_POSITION {
	return float4(positionOS, 1.0);
}
```

我们同样需要给输入值添加语义，因为顶点数据不止包含了坐标。现在这种情况我们需要 POSITION 添加在参数名后面。

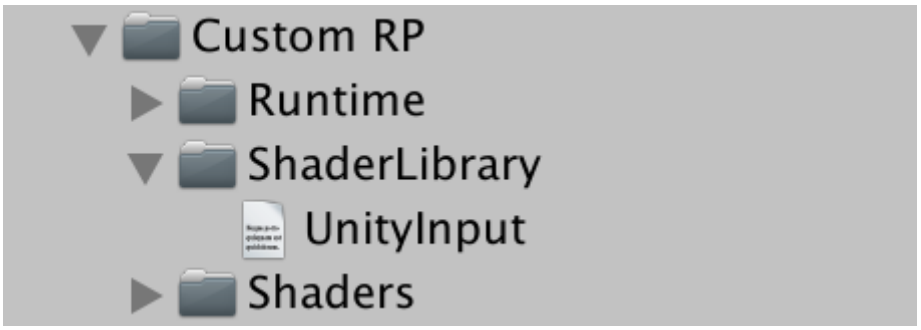
```
float4 UnlitPassVertex (float4 positionOS : POSITION) : SV_POSITION {
	return float4(positionOS, 1.0);
}
```



使用物体空间坐标

网格再次显现出来，但是不正确，因为我们输出的坐标在错误的空间里。空间转换需要用到矩阵，它是在绘制物体的时候发送给 GPU 的。我们必须在着色器中添加这些矩阵，但由于

他们始终一样，所以我们将 Unity 提供的标准输入放在一个单独的 HLSL 文件里，这样既可以让代码结构化，也使得能够在别的着色器中引用。新建一个 UnityInput.hlsl 文件，并将它放到 Custom RP 下的 ShaderLibrary 文件夹中，参照 Unity 渲染管线的文件夹结构。



ShaderLibrary 文件夹和 UnityInput 文件

在文件开头用 CUSTOM_UNITY_INPUT_INCLUDED 做引用保护，然后在全局作用域中定义一个名为 unity_ObjectToWorld 的 float4x4 矩阵。在 C#的类中，这将定义一个字段，但在这里被称作统一值。它由 GPU 在每次绘制时设置一次，对当次绘制内的所有顶点和片元方法的调用保持不变（统一）。

```
#ifndef CUSTOM_UNITY_INPUT_INCLUDED
#define CUSTOM_UNITY_INPUT_INCLUDED

float4x4 unity_ObjectToWorld;

#endif
```

我们能够使用这个矩阵从物体空间转换到全局空间。因为是个常用功能，所以我们创建一个方法，并且放到另一个文件里，文件命名为 Common.hlsl，同样放在 ShaderLibrary 文件夹下。我们在这里引入 UnityInput 并且声明一个输入和输出都是 float3 类型的 TransformObjectToWorld 方法。

```
#ifndef CUSTOM_COMMON_INCLUDED
#define CUSTOM_COMMON_INCLUDED

#include "UnityInput.hlsl"

float3 TransformObjectToWorld (float3 positionOS) {
	return 0.0;
}

#endif
```

空间变换通过调用矩阵和向量的 mul 方法来完成。目前我们需要一个 4D 向量，但由于第四个分量始终是 1，我们可以使用自己添加的 float4(positions, 1.0)。运算结果依然是一个第四分量始终为 1 的 4D 向量。我们可以通过访问向量的 xyz 属性来提取前三个分量，这被称为 swizzle 操作。

```
float3 TransformObjectToWorld (float3 positionOS) {
    return mul(unity_ObjectToWorld, float4(positionOS, 1.0)).xyz;
}
```

现在我们可以在 UnlitPassVertex 里转换到世界空间了。先在方法上方直接引入 Common.hlsl。由于文件在另一个文件夹，我们可以通过相对路径../ShaderLibrary/Common.hlsl 来获取。然后用 TransformObjectToWorld 方法来计算 positionWS 变量，并将其代替物体空间坐标返回。

```
#include "../ShaderLibrary/Common.hlsl"

float4 UnlitPassVertex (float3 positionOS : POSITION) : SV_POSITION {
    float3 positionWS = TransformObjectToWorld(positionOS.xyz);
    return float4(positionWS, 1.0);
}
```

现在结果依然是错的，因为我们需要的是齐次裁剪空间里的坐标。这个空间定义了一个包含摄像机视野内所有物体的立方体，如果是透视矩阵的话则会变成梯形。从世界空间转换到裁剪空间可以通过乘视图投影矩阵来实现，该矩阵包含了摄像机位置，方向，投影，视场角和近-远裁剪平面。可以通过 unity_MatrixVP 矩阵来获取，所以把它添加到 UnityInput.hlsl 里。

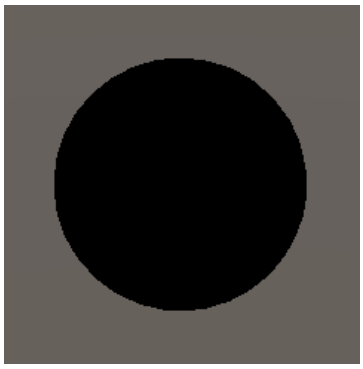
```
float4x4 unity_ObjectToWorld;

float4x4 unity_MatrixVP;
```

在 Common.hlsl 里添加一个与 TransformObjectToWorld 相同的 TransformWorldToHClip 方法，输入改成世界空间坐标，使用 unity_MatrixVP 矩阵，并且输出一个 float4。

```
float3 TransformObjectToWorld (float3 positionOS) {
    return mul(unity_ObjectToWorld, float4(positionOS, 1.0)).xyz;
}

float4 TransformWorldToHClip (float3 positionWS) {
    return mul(unity_MatrixVP, float4(positionWS, 1.0));
}
```

让 UnlitPassVertex 使用该方法在正确的空间中返回坐标。

```
float4 UnlitPassVertex (float3 positionOS : POSITION) : SV_POSITION {
    float3 positionWS = TransformObjectToWorld(positionOS.xyz);
    return TransformWorldToHClip(positionWS);
}
```



正确的黑色球体

1.6. 核心库

刚才我们定义的两个方法非常常用，所以他们也包含在 Core RP Pipeline 里。这个核心库定义了许多更有用和必须的内容，所以我们可以安装这个包，并且移除我们自己的方法定义，然后用库中相关文件代替，这里用到的文件是 Packages/com.unity.render-pipelines.core/ShaderLibrary/SpaceTransforms.hlsl。

```
1.  //float3 TransformObjectToWorld (float3 positionOS) {
2.  //  return mul(unity_ObjectToWorld, float4(positionOS, 1.0)).xyz;
3.  //}
4.
5.  //float4 TransformWorldToHClip (float3 positionWS) {
6.  //  return mul(unity_MatrixVP, float4(positionWS, 1.0));
7.  //}
8.
9.  #include "Packages/com.unity.render-
    pipelines.core/ShaderLibrary/SpaceTransforms.hlsl"
```

这样会编译失败，因为 SpaceTransforms.hlsl 里的代码不存在 unity_ObjectToWorld，而是将相关矩阵用宏 UNITY_MATRIX_M 来定义，所以我们在引入文件之前加入一行#define UNITY_MATRIX_M unity_ObjectToWorld 宏定义。在这之后，所用出现 UNITY_MATRIX_M 的地方都会被替换成 unity_ObjectToWorld。我们稍后再解释这么做的原因。

```
1.  #define UNITY_MATRIX_M unity_ObjectToWorld
2.
3.  #include "Packages/com.unity.render-
    pipelines.core/ShaderLibrary/SpaceTransforms.hlsl"
```

逆矩阵 unity_WorldToObject 也是如此，对应 UNITY_MATRIX_I_M，unity_MatrixV 矩阵对应 UNITY_MATRIX_V, unity_MatrixVP 对应 UNITY_MATRIX_VP。最后，还有通过 UNITY_MATRIX_P 定义的投影矩阵，可以作为 glstate_matrix_projection 使用。我们不需要这些额外的矩阵，但如果没有包含他们，代码无法编译。

```
#define UNITY_MATRIX_M unity_ObjectToWorld
#define UNITY_MATRIX_I_M unity_WorldToObject
#define UNITY_MATRIX_V unity_MatrixV
#define UNITY_MATRIX_VP unity_MatrixVP
#define UNITY_MATRIX_P glstate_matrix_projection
```

同样将这些额外的矩阵添加到 UnityInput 里。

```
float4x4 unity_ObjectToWorld;
float4x4 unity_WorldToObject;

float4x4 unity_MatrixVP;
float4x4 unity_MatrixV;
float4x4 glstate_matrix_projection;
```

除了矩阵之外还最后缺少一个 unity_WorldTransformParams，也包含了我们这里不需要的转换信息。它是一个 real4 向量，这本身不是一个合法类型，取决于在不同目标平台，作为 float4 或 half4 的别名。

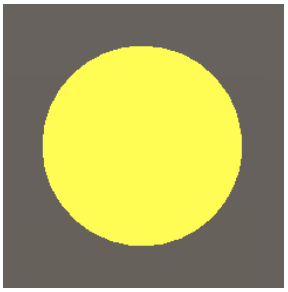
```
float4x4 unity_ObjectToWorld;
float4x4 unity_WorldToObject;
real4 unity_WorldTransformParams;
```

这个别名和很多其他的基础宏在每个图形 API 中都有定义，我们可以通过引入 Packages/com.unity.render-pipilines.core/ShaderLibrary/Common.hlsl 来获取。在 Common.hlsl 里引入 UnityInput.hlsl 之前引入。如果对这些文件感兴趣，你可以在导入的包里查看这些文件。

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

1.7. 颜色

被渲染物体的颜色可以通过调整 UnlitPassFragment 来改变。例如，我们可以通过把返回值从 0 改为 flota4(1.0, 1.0, 0.0, 1.0)来将其改成黄色。



黄色球体

为了实现对每个材质配置颜色，我们必须将其重新定义为一个统一值。在引入指令和 UnlitPassVertex 方法之间进行定义。我们需要一个 float4 然后将其命名为_BaseColor。下划线开头是表明它代表材质属性的标准写法。在 UnlitPassFragment 里用它替代一个硬编码的颜色作为返回值。

```
#include "../ShaderLibrary/Common.hlsl"

float4 _BaseColor;

float4 UnlitPassVertex (float3 positionOS : POSITION) : SV_POSITION {
	float3 positionWS = TransformObjectToWorld(positionOS);
	return TransformWorldToHClip(positionWS);
}

float4 UnlitPassFragment () : SV_TARGET {
	return _BaseColor;
}
```

现在变回了黑色，因为默认值是 0。为了与材质连接起来，我们必须在 Unlit 着色器文件的 Properties 块里添加_BaseColor。

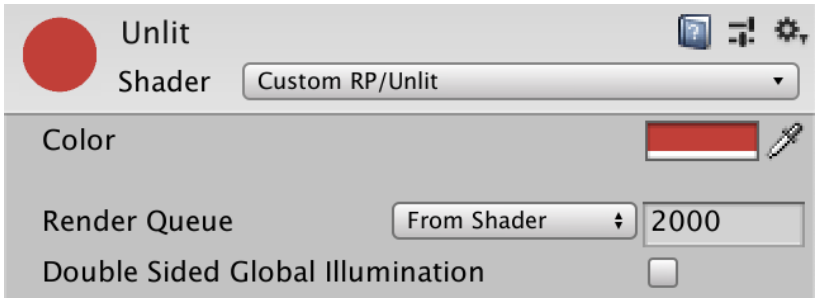
```
Properties {
	_BaseColor
}
```

属性名必须后面带一个面板中使用的字符串和一个 Color 类型声明，就像为方法提供参数一样。

```
_BaseColor("Color", Color)
```

最后，我们必须提供一个默认值，这里需要分配一个 4 个数字的列表，我们使用白色。

```
	_BaseColor("Color", Color) = (1.0, 1.0, 1.0, 1.0)
```



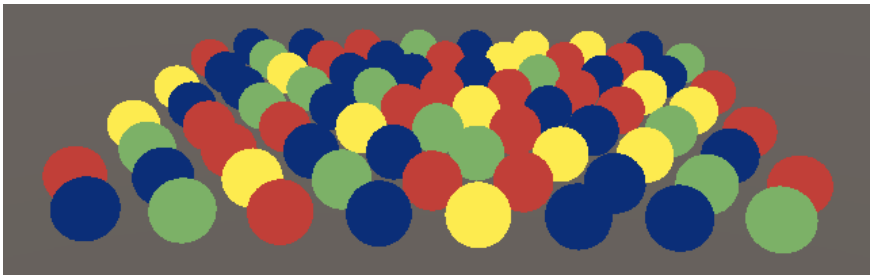
红色的不受光材质

现在可以用我们的着色器创建多个不同颜色的材质了。

2. 批处理

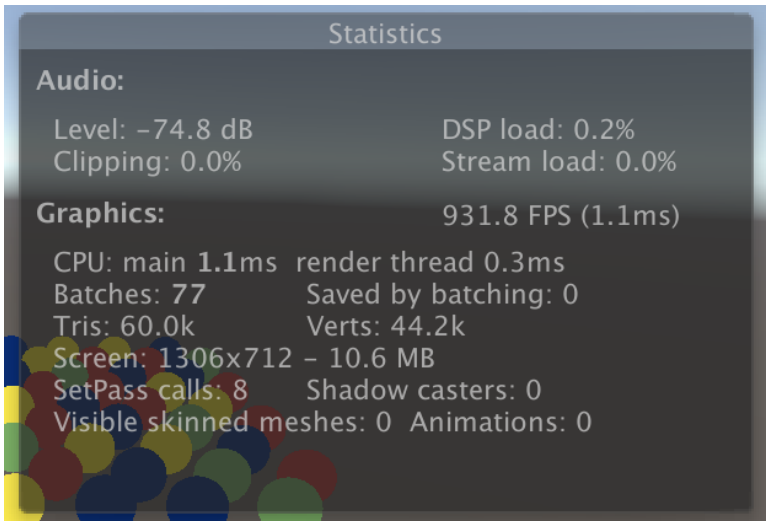
每一次绘制都需要 CPU 和 GPU 之间的通讯。如果有大量数据需要发送给 GPU，可能会因为等待而导致浪费时间，并且当 CPU 忙于发送数据时，它无法处理其他事情。这两个问题都会导致帧率下降。目前我们的方法很直接：每个物体执行自己的绘制。这是最差的方法，因为我们只发送了很少的数据，所以现在看起来还行。

举个例子，我在一个场景中放了 76 个球，这些球使用了红绿黄蓝四种材质之一。它需要 78 次绘制来进行渲染，其中 76 次是球体的，一次是天空盒，还有一次是清理渲染目标。



76 个球体，78 次绘制

如果你打开 Game 窗口的 Stats 面板，就可以看到当前帧渲染的概况。有趣的一点是它显示了 77 次绘制（忽略了清理），批处理节省的次数为 0。

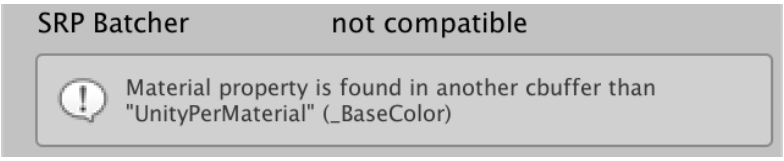


Game 窗口的统计

2.1 SRP Batcher

合批是合并绘制调用的过程，它可以减少 CPU 和 GPU 之间通讯消耗的时间。合批最简单的方法是开启 SRP Batcher，但这只对兼容的着色器有效，而我们的 Unlit 着色器不符合。你可

以在监视面板里确诊这点。有一行 SRP Batcher 表明不兼容，在下面给出了具体原因。



不兼容

SRP Batcher 只是让绘制变得更精简，而非减少绘制数量。它将材质属性缓存在 GPU 中，所以不需要每次绘制都上传。这既减少了必须上传的数据量，也减少了每次绘制 CPU 的工作量。但这只有在着色器遵循严格的结构来获取统一数据时才有效。

所有的材质属性都必须在一个特定的内存缓冲区里定义，而不是在全局层面。这需要将 _BaseColor 声明包含在一个命名为 UnityPerMaterial 的 cbuffer 块中。这类似于一个结构定义，但必须使用分号结尾。它通过将_BaseColor 放入特定的常量缓冲区来进行隔离，尽管它在全局层面依然可用。

```
cbuffer UnityPerMaterial {
	float _BaseColor;
};
```

并非所有平台都支持常量缓冲区（例如 OpenGL ES 2.0），所以我们需要使用 Core RP 库中的 CBUFFER_START 和 CBUFFER_END 宏来代替直接使用 cbuffer。CBUFFER_START 像一个方法一样接受缓冲区名称作为参数。目前我们得到了和之前完全一样的情况，只是对于不支持 cbuffer 的平台不会存在 cbuffer 代码。

```
CBUFFER_START(UnityPerMaterial)
	float4 _BaseColor;
CBUFFER_END
```

对于 unity_ObjectToWorld，unity_WorldToObject 和 unity_WorldTransformParams 也要同样处理，只不过把它们归类在 UnityPerDraw 缓冲区里。

```
CBUFFER_START(UnityPerDraw)
	float4x4 unity_ObjectToWorld;
	float4x4 unity_WorldToObject;
	real4 unity_WorldTransformParams;
CBUFFER_END
```

在这种情况下，如果我们需要使用特定组中的某一个数值，就需要定义整个组的数据。对于变换组来说，我们还需要包含一个 float4 unity_LODFade，尽管我们用不到。先后顺序不重要，但 Unity 将它放在了 unity_WorldToObject 之后，所以我们也照做。

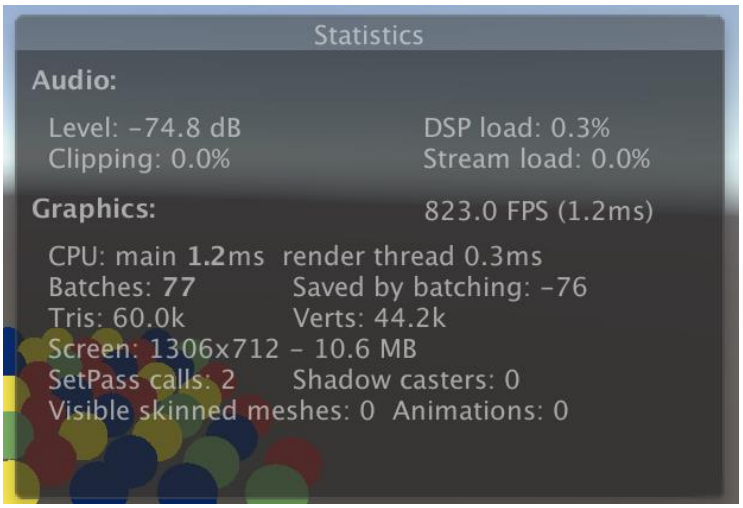
```
CBUFFER_START(UnityPerDraw)
	float4x4 unity_ObjectToWorld;
	float4x4 unity_WorldToObject;
```

```
    float4 unity_LODFade;
    real4 unity_WorldTransformParams;
CBUFFER_END
```

兼容 SRP Batcher

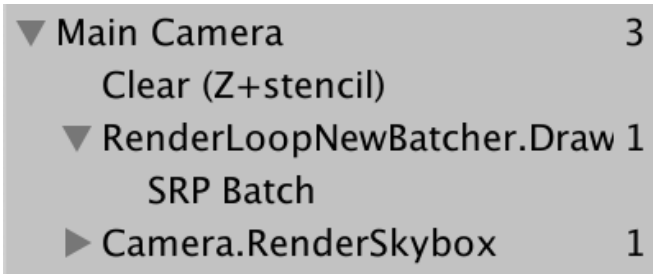
着色器兼容之后，下一步是开启 SRP Batcher，这需要通过将 GraphicsSettings.useScriptableRenderPipelineBatching 设置成 true 来实现。我们只需要设置一次，所以我们新增一个 CustomRenderPipeline 构造函数，在管线实例创建的时候进行操作。

```
public CustomRenderPipeline () {
    GraphicsSettings.useScriptableRenderPipelineBatching = true;
}
```



减少批次

统计面板显示节省了 76 个批次，虽然它显示为负值。帧调试器现在显示在 RenderLoopNewBatcher.Draw 下面只有一个 SRP Batch 条目，然而需要注意的是它并不是一次绘制，而是多次绘制的优化序列。



一个 SRP Batch

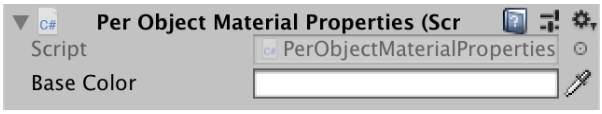
2.2. 多颜色

尽管我们使用了四种材质，但只得到了一个批次。因为它们的数据被缓存在 GPU 上，并且每一次绘制只需要包含一个正确内存地址的偏移量。唯一的限制是每个材质的内存布局必须一致，这正是因为我们给所有材质使用了同样的着色器，其中只包含一个颜色属性。Unity 不会去对比材质精确的内存布局，只是简单的对使用完全相同的着色器变体的绘制进行批次合并。

如果我们只想要少数几种颜色，这种方法是可行的，但是如果我们想给每个球体单独的颜色，就需要创建更多的材质。如果我们可以改为对每个物体设置颜色就更方便了。这在默认情况下无法实现，但我们可以通过创建一个自定义组件类型来支持这种情形，将其命名为 PerObjectMaterialProperties。因为它是一个示例，所以我把它放在 Custom RP 下的 Examples 文件夹里。

这个想法是，一个 Game Object 可以有一个与其关联的 PerObjectMaterialProperties 组件，它含有一个 Base Color 的配置选项，可以用于设置材质的_BaseColor 属性。它需要知道着色器属性的标签 ID，我们可以通过 Shader.PropertyToID 来获取，并将其存为静态变量，类似于对 CameraRenderer 里着色器 pass 标签 ID 的处理方式，尽管在这里变量是一个整数。

```
1.  using UnityEngine;
2.
3.  [DisallowMultipleComponent]
4.  public class PerObjectMaterialProperties : MonoBehaviour {
5.
6.      static int baseColorId = Shader.PropertyToID("_BaseColor");
7.
8.      [SerializeField]
9.      Color baseColor = Color.white;
10. }
```



PerObjectMaterialProperties 组件

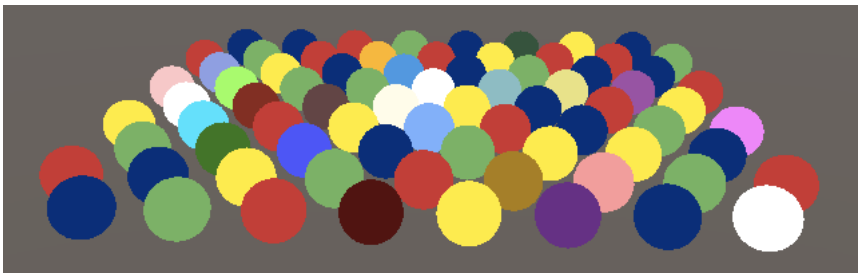
通过 MaterialPropertyBlock 给每个物体设置材质属性。我们只需要一个实例，因为所有的 PerObjectMaterialProperties 实例都可以复用，所以声明一个静态字段。

```
1.  static MaterialPropertyBlock block;
```

如果不存在，新建一个 block，然后调用 SetColor 传入属性 ID 和颜色，再通过 SetPropertyBlock 方法将该 block 应用到物件的 Renderer 组件上，用来复制设置。在 OnValidate 中执行，以便可以立刻在编辑器种看到结果。

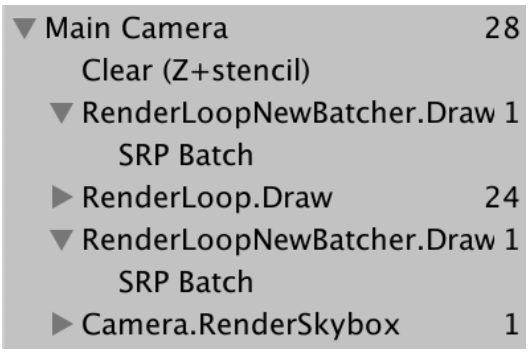
```
void OnValidate () {
    if (block == null) {
        block = new MaterialPropertyBlock();
    }
    block.SetColor(baseColorId, baseColor);
    GetComponent<Renderer>().SetPropertyBlock(block);
}
```

我给 24 个任意的球体添加了这个组件，并给与他们不同的颜色。



许多颜色

不幸的是 SRP Batcher 不能处理每个物体的材质属性，所以这 24 个球变回了每个进行一次绘制，并且由于排序的影响，可能将其他球体也拆分为多个批次。



24 个未被合批的绘制

另外，OnValidate 在构建版本中不会被调用，为了显示各自的颜色，我们必须在 Awake 里也进行一次操作，可以通过简单的调用 OnValidate 实现。

```
void Awake () {
    OnValidate();
}
```

2.3. GPU 实例化

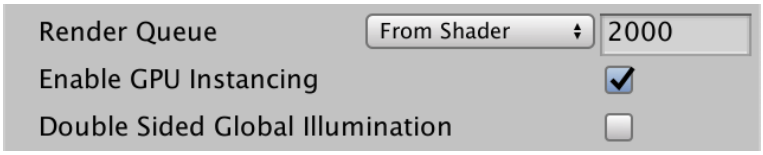
还有一种合并绘制的方法，它对每个对象材质属性有效。它被称作 GPU 实例化，通过将使用相同网格的对象通过一次绘制调用来实现。CPU 获取每个对象的变换数据和材质属性，然

后将它们放在数组中发送给 GPU。然后 GPU 遍历所有条目并按照提供的顺序来渲染它们。

因为 GPU 实例化需要提供数组类型的数据，我们的着色器目前还不支持。第一步是在 Pass 块的顶点和片元 pragma 前面加上#pragma multi_compile_instancing 指令。

```
1.  #pragma multi_compile_instancing
2.  #pragma vertex UnlitPassVertex
3.  #pragma fragment UnlitPassFragment
```

这样会使得 Unity 为我们的着色器生成两个变体，一个支持 GPU 实例化，另一个不支持。材质面板中也多了一个开关选项，是我们可以对每个材质选择使用哪个版本。



开启 GPU 实例化的材质

支持 GPU 实例化需要改变方法，我们必须从核心着色器库中引入 UnityInstancing.hlsl 文件。这步必须在 UNITY_MATRIX_M 和其他宏定义之后和引入 SpaceTransforms.hlsl 之前。

```
1.  #define UNITY_MATRIX_P glstate_matrix_projection
2.
3.  #include "Packages/com.unity.render-
    pipelines.core/ShaderLibrary/UnityInstancing.hlsl"
4.  #include "Packages/com.unity.render-
    pipelines.core/ShaderLibrary/SpaceTransforms.hlsl"
```

UnityInstancing.hlsl 的作用是重新定义了宏用来访问实例数据数组。但为了使其有效，着色器需要知道正在渲染的对象索引。索引值通过顶点数据提供，所以我们可以获得。UnityInstancing.hlsl 定义了宏来简化这个过程，但它们默认我们的定点方法使用结构体参数。

可以像 cbuffer 一样声明一个 struct，然后将其作为方法的输入参数。我们同样可以在结构体内定义语义。这种写法的优势在于比一长串的参数列表要更清晰。所以将 UnlitPassVertex 的参数包含在 Attributes 结构中，作为顶点输入数据。

```
1.  struct Attributes {
2.      float3 positionOS : POSITION;
3.  };
4.
5.  float4 UnlitPassVertex (Attributes input) : SV_POSITION {
6.      float3 positionWS = TransformObjectToWorld(input.positionOS);
7.      return TransformWorldToHClip(positionWS);
8.  }
```

当使用 GPU 实例化时，对象索引也可以作为顶点属性使用。我们可以在 Attributes 里加入

UNITY_VERTEX_INPUT_INSTANCE_ID 来使用。

```
struct Attributes {
    float3 positionOS : POSITION;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

接下来，在 UnlitPassVertex 开头添加 UNITY_SETUP_INSTANCE_ID(input);，它从 input 中提取索引，并将其存在其他实例化宏依赖的全局静态变量里。

```
float4 UnlitPassVertex (Attributes input) : SV_POSITION {
    UNITY_SETUP_INSTANCE_ID(input);
    float3 positionWS = TransformObjectToWorld(input.positionOS);
    return TransformWorldToHClip(positionWS);
}
```

这样一来 GPU 实例化就可以生效了，尽管由于 SRP Batcher 优先级更高，所以结果上没有变化。但我们还没有支持每个实例的材质数据。为了添加这个功能，我们必须在需要的时候把 _BaseColor 改成数组引用。这通过将 CBUFFER_START 和 CBUFFER_END 改成 UNITY_INSTANCING_BUFFER_START 和 UNITY_INSTANCING_BUFFER_END 来实现，其中 End 同样需要一个参数。这个参数不需要与开始的一样，但没有什么特别的原因来使用不同的参数。

```
//CBUFFER_START(UnityPerMaterial)
//  float4 _BaseColor;
//CBUFFER_END

UNITY_INSTANCING_BUFFER_START(UnityPerMaterial)
    float4 _BaseColor;
UNITY_INSTANCING_BUFFER_END(UnityPerMaterial)
```

然后将_BaseColor 的定义换成 UNITY_DEFINE_INSTANCED_PROP(float4, _BaseColor)。

```
UNITY_INSTANCING_BUFFER_START(UnityPerMaterial)
    //  float4 _BaseColor;
    UNITY_DEFINE_INSTANCED_PROP(float4, _BaseColor)
UNITY_INSTANCING_BUFFER_END(UnityPerMaterial)
```

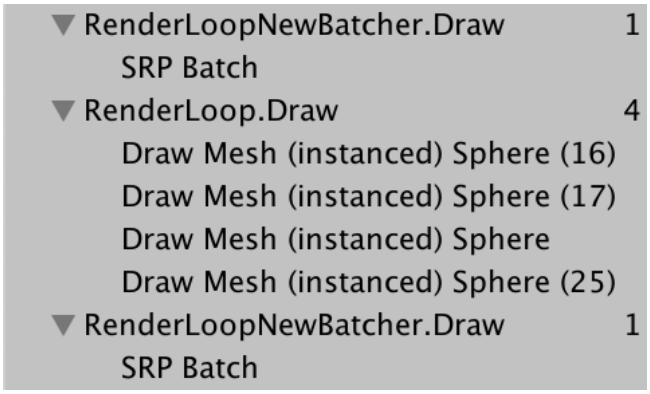
当使用实例化时，我们还必须让实例化索引在 UnlitPassFragment 中同样可用。为了简化流程，我们使用一个同时包含位置和索引的结构体作为 UnlitPassVertex 的输出。当索引存在时，使用 UNITY_TRANSFER_INSTANCE_ID(input, output);来复制它。我们像 Unity 一样把这个结构体命名为 Varyings，因为它包含的数据可以在同一个三角面的片元之间变换。

```
struct Varyings {
    float4 positionCS : SV_POSITION;
```

```
    UNITY_VERTEX_INPUT_INSTANCE_ID
};

Varyings UnlitPassVertex (Attributes input) { //: SV_POSITION {
    Varyings output;
    UNITY_SETUP_INSTANCE_ID(input);
    UNITY_TRANSFER_INSTANCE_ID(input, output);
    float3 positionWS = TransformObjectToWorld(input.positionOS);
    output.positionCS = TransformWorldToHClip(positionWS);
    return output;
}
```

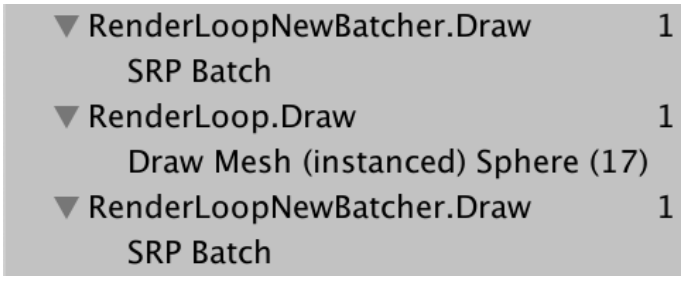
将这个结构体作为 UnlitPassFragment 的参数。然后像之前一样，使用 UNITY_SETUP_INSTANCE_ID 来让索引可用。材质属性现在必须通过 UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, _BaseColor)来访问。

```
float4 UnlitPassFragment (Varyings input) : SV_TARGET {
    UNITY_SETUP_INSTANCE_ID(input);
    return UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, _BaseColor);
}
```



实例化绘制调用

Unity 现在可以将 24 个带有每个对象颜色的球体合并，来减少绘制调用的次数。我最终使用了四个实例化绘制调用，因为这些球体依然使用了四种材质。GPU 实例化只对共享了相同材质的对象起效。尽管它们重写了材质颜色，但依然是同一个材质，这使得他们可以在一个批次里绘制。



一个实例化材质

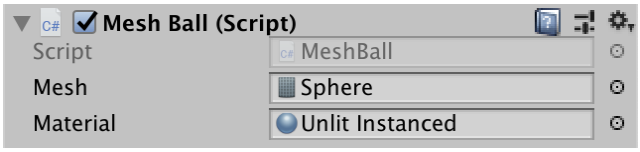
需要注意的是，对于批次的大小是有限制的，这基于目标平台和每个实例需要提供多少数据。如果超出了限制，你就会得到不止一个批次。除此之外，如果使用了多种材质，排序也会导致批次拆分。

2.4. 绘制多实例网格

当上百个对象可以合并在一次绘制中时，GPU 实例化就变得效果显著。但是手动编辑场景中这么多对象不太实际。所以让我们随机生成一堆。创建一个 MeshBall 示例组件用于在激活时生成许多对象。让它缓存着色器属性_BaseColor 并且分别添加一个网格选项和材质选项，该材质必须支持实例化。

```
1.  using UnityEngine;
2.
3.  public class MeshBall : MonoBehaviour {
4.
5.      static int baseColorId = Shader.PropertyToID("_BaseColor");
6.
7.      [SerializeField]
8.      Mesh mesh = default;
9.
10.     [SerializeField]
11.     Material material = default;
12. }
```

创建一个带有该组件的游戏对象。我给它添加了默认的球体网格用来绘制。



球体 Mesh Ball 组件

我们可以创建许多新的游戏对象，但是没必要。相反，我们用变换矩阵和颜色来填充一个数组，然后通知 GPU 用它来渲染网格。这是 GPU 实例化最有用的地方。我们可以在一次操作中提供高达 1023 个实例，所以我们添加对应长度的数组字段，再加一个 MaterialPropertyBlock 用来传递颜色数据。在这种情况下，颜色数组的元素类型必须是 Vector4。

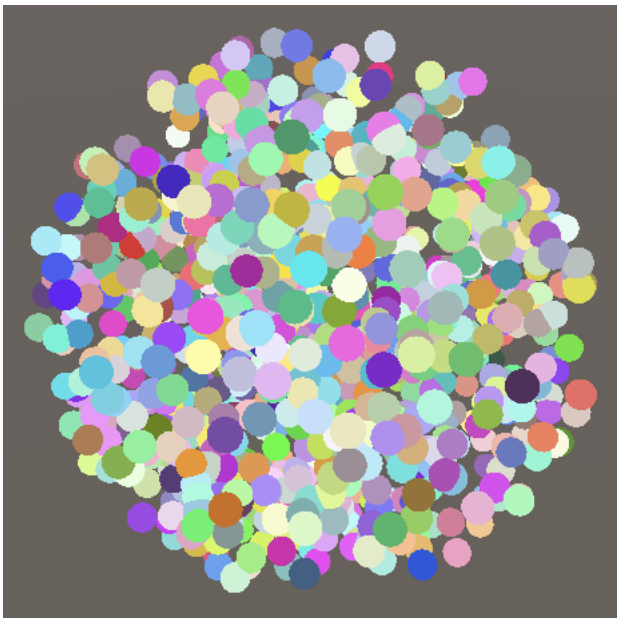
```
1.  Matrix4x4[] matrices = new Matrix4x4[1023];
2.  Vector4[] baseColors = new Vector4[1023];
3.
4.  MaterialPropertyBlock block;
```

创建一个 Awake 方法，然后用在半径 10 以内的随机点和随机 RGB 颜色数据来填充这些数组。

```
void Awake () {
	for (int i = 0; i < matrices.Length; i++) {
		matrices[i] = Matrix4x4.TRS(
			Random.insideUnitSphere * 10f, Quaternion.identity, Vector3.one

		);
		baseColors[i] =
			new Vector4(Random.value, Random.value, Random.value, 1f);
	}
}
```

在 Update 里，如果 block 不存在，则创建一个新 block，然后调用 SetVectorArray 来配置颜色。在那之后，调用 Graphics.DrawMeshInstanced 传入 mesh，子网格索引 0，material，matrices 数组，元素个数和属性 block 作为参数。我们在这里设置 block 以便网格球避免热重载。

```
void Update () {
	if (block == null) {
		block = new MaterialPropertyBlock();
		block.SetVectorArray(baseColorId, baseColors);
	}
	Graphics.DrawMeshInstanced(mesh, 0, material, matrices, 1023, block);
}
```



1023 个球，3 次绘制

进入 play mode 之后会产生一个密集的球体区域。消耗几次绘制取决于平台，因为每次绘制使用的最大缓冲区大小不同。目前它用了三次绘制来渲染。

需要注意的是，这些独立的网格是基于与我们提供的数据的相同顺序来绘制的。除此之外，

没有进行任何排序或者剔除，尽管当它在视锥外时，真个批次都会消失。

2.5. 动态合批

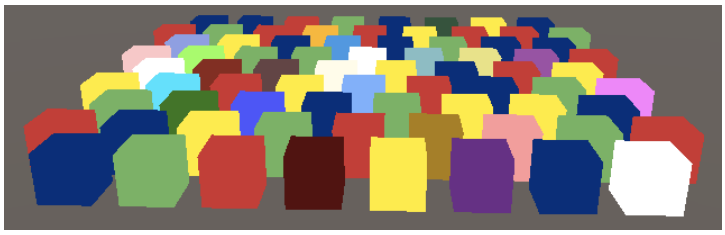
动态合批是第三种可以减少绘制调用的方法。这是一种比较旧技术，可以将使用相同材质的多个小网格合并为一个单独的大网格进行绘制。当使用独立对象材质属性时同样无法生效。

较大的网格是按需生成的，因此只适用于合并较小的网格。球体网格太大，但立方体可以。为了在实际中看到效果，我们禁用 GPU 实例化，并在 CameraRenderer.DrawVisibleGeometry 里把 enableDynamicBatching 设置为 true。

```
1.  var drawingSettings = new DrawingSettings(
2.      unlitShaderTagId, sortingSettings
3.  ) {
4.      enableDynamicBatching = true,
5.      enableInstancing = false
6.  };
```

同样禁用 SRP Batcher，因为它的优先级更高。

```
1.      GraphicsSettings.useScriptableRenderPipelineBatching = false;
```



改为绘制立方体

通常来说，GPU 实例化比动态合批更有效。这个方法同样也有些注意事项，例如当缩放不同时，就不能确保大网格的法向量是单位向量。同样的，因为现在是一个单独网格而不是多个，所以绘制顺序也发生了变化。

另外还有静态合批，方法类似，但是只对提前标记为静态批处理的对象进行操作。除了需要更多的内存和存储空间外，没有别的限制。这个渲染管线不考虑，所以我们不用担心。

2.6. 配置批处理

最好的方法可能有所不同，所以我们把它设置成可配置。首先给 DrawVisibleGeometry 添加布尔值参数来控制是否使用动态合批和 GPU 实例化来代替硬编码。

```
void DrawVisibleGeometry (bool useDynamicBatching, bool useGPUInstancing) {

    var sortingSettings = new SortingSettings(camera) {
        criteria = SortingCriteria.CommonOpaque
    };
    var drawingSettings = new DrawingSettings(
        unlitShaderTagId, sortingSettings
    ) {
        enableuseDynamicBatching = useDynamicBatching,
        enableInstancing = useGPUInstancing
    };
    …
}
```

现在 Render 需要提供这个配置，然后以靠渲染管线提供给它。

```
public void Render (
    ScriptableRenderContext context, Camera camera,
    bool useDynamicBatching, bool useGPUInstancing
) {
    …
    DrawVisibleGeometry(useDynamicBatching, useGPUInstancing);
    …
}
```

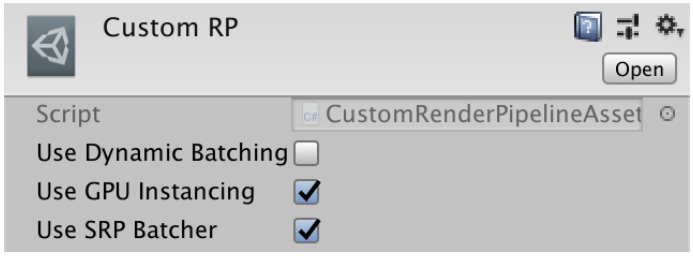
CustomRenderPipeline 可以将这些选项保存为字段，然后在构造函数里设置，并将它们传给 Render。同样添加一个布尔值参数用来控制 SRP Batcher 而不是始终开启。

```
bool useDynamicBatching, useGPUInstancing;

public CustomRenderPipeline (
    bool useDynamicBatching, bool useGPUInstancing, bool useSRPBatcher
) {
    this.useDynamicBatching = useDynamicBatching;
    this.useGPUInstancing = useGPUInstancing;
    GraphicsSettings.useScriptableRenderPipelineBatching = useSRPBatcher;
}

protected override void Render (
    ScriptableRenderContext context, Camera[] cameras
) {
    foreach (Camera camera in cameras) {
        renderer.Render(
            context, camera, useDynamicBatching, useGPUInstancing
        );
```

```
18.     }
19. }
```

最后，在 CustomRenderPipelineAsset 里添加所有的三个参数作为配置字段，将他们传给 CreatePipeline 里的构造函数。

```
1.  [SerializeField]
2.  bool useDynamicBatching = true, useGPUInstancing = true, useSRPBatcher = tru
    e;
3.
4.  protected override RenderPipeline CreatePipeline () {
5.      return new CustomRenderPipeline(
6.          useDynamicBatching, useGPUInstancing, useSRPBatcher
7.      );
8.  }
```



管线配置

现在可以更改我们的渲染管线所用的方法了。勾选一个选项会立刻起效，因为当资产发生变化时，Unity 编辑器会创建一个新的 RP 实例。

3. 透明

我们的着色器可以用来创建不受光的不透明材质。现在可以调整颜色的 alpha 分量，这通常用于表示透明度，但现在没有效果。我们也可以将渲染队列设置成 Transparent，但这只会在对象被绘制的时候改变，并且它只改变绘制顺序，并不改变绘制方式。

减少 alpha 并且使用透明渲染队列

我们不需要写一个单独的着色器来支持透明材质。对我们的 Unlit 着色器进行一点小感动就可以同时支持不透明和透明渲染。

3.1. 混合模式

不透明渲染和透明渲染的主要差异在于我们是替换之前的绘制还是与先前的绘制结果合并以产生透视效果。我们可以通过设置源对象和目标对象的混合模式来控制这一点。这里源对象指的是现在正在绘制，而目标对象是之前绘制的内容和最终结果绘制的目标。为它们添加两个着色器属性：_SrcBlend 和_DstBlend。它们是混合模式的枚举，但我们最好还是用 Float，并默认将_SrcBlend 设为 1，_DstBlend 设为 0。

```
1.  Properties {
2.      _BaseColor("Color", Color) = (1.0, 1.0, 1.0, 1.0)
3.      _SrcBlend ("Src Blend", Float) = 1
4.      _DstBlend ("Dst Blend", Float) = 0
5.  }
```

为了更易于编辑，我们可以将 Enum 属性添加到这些属性里，并使用完全限定的 UnityEngine.Rendering.BlendMode 枚举类型作为参数。

```
1.  [Enum(UnityEngine.Rendering.BlendMode)] _SrcBlend ("Src Blend", Float) = 1
2.  [Enum(UnityEngine.Rendering.BlendMode)] _DstBlend ("Dst Blend", Float) = 0
```

不透明混合模式

默认值代表了我们正在使用的不透明混合配置。源对象设置为 1，意味着被完整添加，同时目标对象设置为 0，意味着被忽略。

对于标准透明对象的源混合模式是 SrcAlpha，意思是渲染颜色的 RGB 分量与自身的 alpha 分量相乘。所以 alpha 越小值越小。目标混合模式正好相反：OneMinusSrcAlpha，是的总和为 1。

透明混合模式

混合模式可以在 Pass 块中用 Blend 声明再加两个模式来定义。我们想使用着色器属性，可以通过将他们放在方括号内来访问它们。这是可编程着色器之前的古老语法。

```
1.  Pass {
2.      Blend [_SrcBlend] [_DstBlend]
3.
4.      HLSLPROGRAM
5.      …
```

```
6.      ENDHLSL
7.  }
```

半透明的黄色球体

3.2. 不写入深度值

透明渲染通常不写入深度缓冲区，因为它不会从中受益，甚至可能产生意料之外的结果。我们可以通过 ZWrite 声明来控制是否写入深度。同样的，我们可以使用一个着色器属性_ZWrite。

```
1.  Blend [_SrcBlend] [_DstBlend]
2.  ZWrite [_ZWrite]
```

用一个自定义的枚举属性 Enum(Off, 0, On, 1)来定义这个着色器属性，这样可以用 0 和 1 创建一个默认值为 1 的开关选项。

```
1.  [Enum(UnityEngine.Rendering.BlendMode)] _SrcBlend ("Src Blend", Float) = 1
2.  [Enum(UnityEngine.Rendering.BlendMode)] _DstBlend ("Dst Blend", Float) = 0
3.  [Enum(Off, 0, On, 1)] _ZWrite ("Z Write", Float) = 1
```

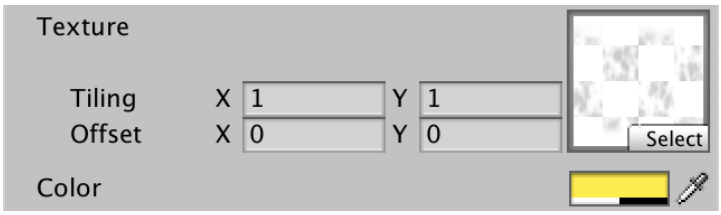
关闭写入深度

3.3. 纹理

之前我们用了一张带 alpha 的纹理来创建非均匀半透明材质。通过给着色器添加一个_BaseMap 纹理属性，我们也可以支持它。这种情况下，类型是 2D，并且我们使用 Unity 的

标准白色纹理作为默认值，这通过字符串 white 来表明。另外 i，我们需要用一个空代码块作为纹理属性的结尾。这以前被用于控制纹理设置，但是现在也需要包含，以免在某些情况下出现奇怪的错误。

```
_BaseMap("Texture", 2D) = "white" {}
_BaseColor("Color", Color) = (1.0, 1.0, 1.0, 1.0)
```



带有纹理的材质

纹理必须上传到 GPU 内存中，这由 Unity 帮我们完成。着色器需要一个相关纹理的句柄，我们可以像定义 uniform 值一样定义它，除了我们使用一个带名称作为参数的宏 TEXTURE2D。我们同样需要定义一个纹理采样状态，用来控制采样方式，包括 wrap mode 和 filter mode。这通过 SAMPLER 宏来实现，类似 TEXTURE2D 但名字前面带有 sampler。这将自动匹配 Unity 提供的采样状态。

纹理和采样器是着色器资源，他们不能基于每个实例提供，并且宾虚在全局作用域中声明。在 UnlitPass.hlsl 的着色器属性之前定义它们。

```
TEXTURE2D(_BaseMap);
SAMPLER(sampler_BaseMap);

UNITY_INSTANCING_BUFFER_START(UnityPerMaterial)
    UNITY_DEFINE_INSTANCED_PROP(float4, _BaseColor)
UNITY_INSTANCING_BUFFER_END(UnityPerMaterial)
```

除此之外，Unity 可以通过与纹理名称后加_ST 的 float4 属性来获取纹理平铺和偏移数据，这些数据代表缩放和变换或其他类似的内容。这个属性应该是 UnityPerMaterial 缓冲区的一部分，所以可以基于每个实例设置。

```
UNITY_INSTANCING_BUFFER_START(UnityPerMaterial)
    UNITY_DEFINE_INSTANCED_PROP(float4, _BaseMap_ST)
    UNITY_DEFINE_INSTANCED_PROP(float4, _BaseColor)
UNITY_INSTANCING_BUFFER_END(UnityPerMaterial)
```

为了对纹理进行采样我们需要纹理坐标，它是顶点属性的一部分。特别地，我们需要第一对坐标，因为坐标可能不止一对。向 Attributes 里添加一个带有 TEXCOORD0 语义的 float2 字段。因为用于我们的基础贴图，并且纹理空间尺寸通常命名为 U 和 V，所以我们将其命名为 baseUV。

```
struct Attributes {
    float3 positionOS : POSITION;
```

```
    float2 baseUV : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

我们需要将坐标传到片元方法中，因为纹理是在那里采样的。所以也在 Varyings 里添加 float2 baseUV。这里我们不需要添加一个特别的语义，它只是我们传递的数据，不需要 GPU 的特别关注。然后，我们仍然给它关联一些语义。我们可以使用任何没有用到的 ID，我们使用 VAR_BASE_UV。

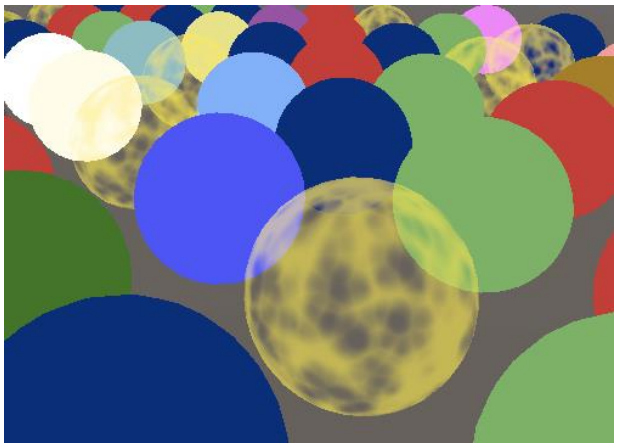
```
struct Varyings {
    float4 positionCS : SV_POSITION;
    float2 baseUV : VAR_BASE_UV;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

当在 UnlitPassVertex 里复制坐标时，我们也可以使用存在_BaseMap_ST 里的缩放和偏移。我们在每个顶点中操作，而不是在片元中。缩放存在 XY 分量，同时偏移存在 ZW 分量，我们可以通过 swizzle 属性访问。

```
Varyings UnlitPassVertex (Attributes input) {
    …

    float4 baseST = UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, _BaseMap_ST);
    output.baseUV = input.baseUV * baseST.xy + baseST.zw;
    return output;
}
```

现在经过三角面内插之后的 UV 坐标在 UnlitPassFragment 中可用了。使用 SAMPLE_TEXTURE2D 宏传入纹理，采样器和坐标作为参数来进行纹理采样。最终的颜色是纯色和纹理的乘法叠加。两个相同长度的向量相乘意味着所有对应的分量相乘，所以在这种情况下，红色乘红色，绿色乘绿色等等。

```
float4 UnlitPassFragment (Varyings input) : SV_TARGET {
    UNITY_SETUP_INSTANCE_ID(input);
    float4 baseMap = SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, input.baseUV);
    float4 baseColor = UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, _BaseColor);
    return baseMap * baseColor;
}
```



带有纹理的黄色球体

因为我们的纹理 RGB 数据是纯白，所以颜色不受影响。但 alpha 通道变了，所以透明度不再是均匀的。

3.4. 透明镂空

另一种透视表面的方法是镂空。着色器也能够做到，这通过丢弃原本一些需要正常渲染的片元来实现。这样会产生硬边缘，而不是我们现在看到的平滑过渡。这个技术被称作透明裁剪（Alpha Clipping）。通常的做法是定义一个裁剪阈值。带有低于阈值的 alpha 值的片元会被丢弃，同时其他的会保留。

添加一个_Cutoff 属性，默认值设为 0.5。因为 alpha 总在 0 和 1 之间，我们可以使用 Rnage(0.0, 1.0)作为该值的类型。

```
1.  _BaseColor("Color", Color) = (1.0, 1.0, 1.0, 1.0)
2.  _Cutoff ("Alpha Cutoff", Range(0.0, 1.0)) = 0.5
```

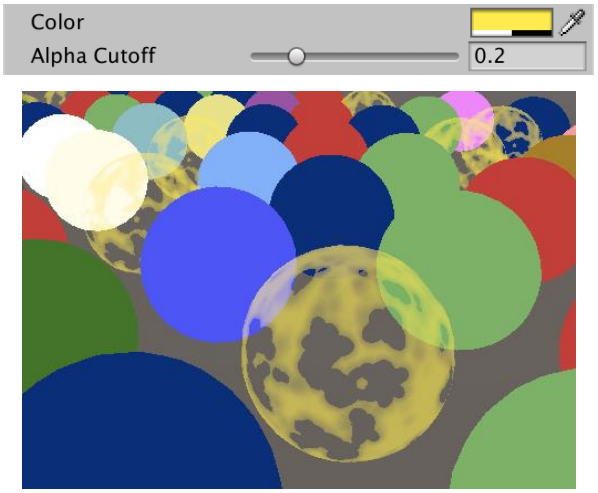
同样把它添加到 UnlitPass 的材质属性里。

```
1.  UNITY_DEFINE_INSTANCED_PROP(float4, _BaseColor)
2.  UNITY_DEFINE_INSTANCED_PROP(float, _Cutoff)
```

我们可以在 UnlitPassFragment 里通过调用 cllip 方法来丢弃片元。如果我们传入的值小于等于 0，它就会中止并丢弃片元。所以将最终的 alpha 值（通过访问 a 或 w 属性）减去裁剪阈值作为传递参数。

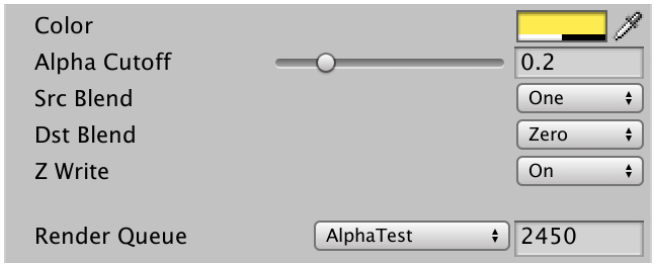
```
1.  float4 baseMap = SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, input.baseUV);

2.  float4 baseColor = UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, _BaseColor)
    ;
3.  float4 base = baseMap * baseColor;
```

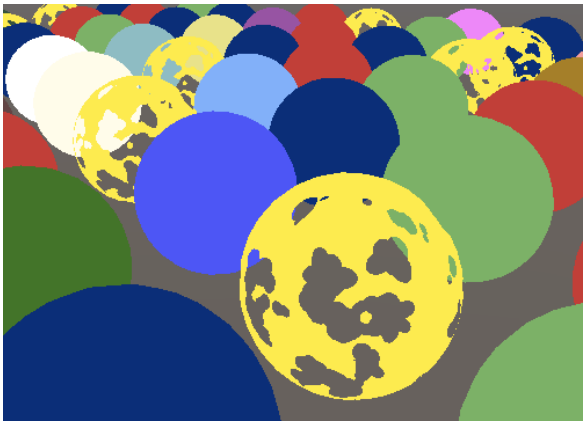
```
4.  clip(base.a - UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, _Cutoff));
5.  return base;
```



Alpha 裁剪设置为 0.2

通常来说，一个材质要么使用透明混合，要么使用 alpha 裁剪，不会同时使用。对一个典型的裁剪材质来说，除了被舍弃的片元以外是完全不透明的，并且会写入深度缓冲。它使用 AlphaTest 渲染队列，意味着它在所有完全不透明对象之后渲染。这么做是因为舍弃片元是的一些 GPU 优化无效，因为不再能假设三角面可以完全挡住它们后面的对象。通过先绘制不透明对象，他们可能会覆盖掉 alpha 裁剪对象的一部分，接下来就不用去处理他们的隐藏片元。





Alpha 镂空材质

但为了执行优化，我们必须确认 clip 只在需要的时候才使用。我们通过添加一个功能开关的

着色器属性来实现。它是一个默认值为 0 的 float 属性，带有一个 Toggle 属性用来控制着色器关键字_CLIPPING。属性本身名称不重要，所以简单命名为_Clipping。

```
1.  _Cutoff ("Alpha Cutoff", Range(0.0, 1.0)) = 0.5
2.  [Toggle(_CLIPPING)] _Clipping ("Alpha Clipping", Float) = 0
```

被认为关闭了 Alpha 镂空

3.5. 着色器功能

打开开关会添加一个_CLIPPING 关键字到材质的生效关键字列表中，同时关闭的时候会移除。但这并没有做任何实行。我们必须告诉 Unity 基于这个关键词是否定义来给我们的着色器编译不同的变笨。我们通过在 Pass 里添加#pragma shader_feature _CLIPPING 来实现这个功能。

```
1.  #pragma shader_feature _CLIPPING
2.  #pragma multi_compile_instancing
```

现在 Unity 会根据_CLIPPING 是否等一来编译我们的着色器代码。它会生成一个或两个变体，这取决于我们怎样配置我们的材质。所以我们可以基于这个定义进行条件判断，类似于引入保护，但目前我们只想在_CLIPPING 定义时才引入 clip 行代码。我们可以使用#ifdef _CLIPPING 来判断，但我更倾向于用#if defined(_CLIPPING)。

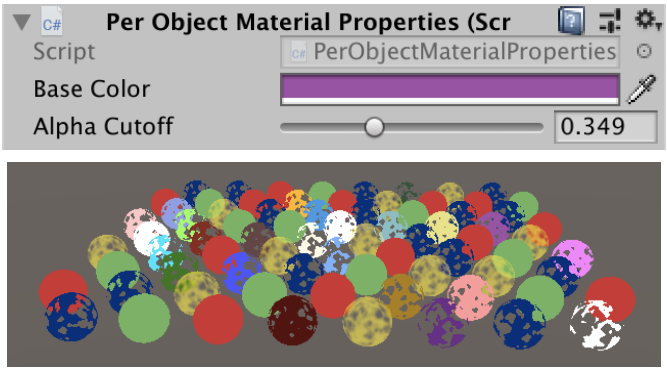
```
1.  #if defined(_CLIPPING)
2.      clip(base.a - UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, _Cutoff));
3.  #endif
```

3.6. 对每个对象进行镂空

因为 cutoff 是 UnityPerMaterial 缓冲区的一部分，它可以对每个实例进行配置。所以我们把这个功能添加到 PerObjectMaterialProperties。它用法和颜色一样，除了我们需要嗲用属性块的 SetFloat 而不是 SetColor。

```
1.  static int baseColorId = Shader.PropertyToID("_BaseColor");
2.  static int cutoffId = Shader.PropertyToID("_Cutoff");
3.
4.  static MaterialPropertyBlock block;
5.
```

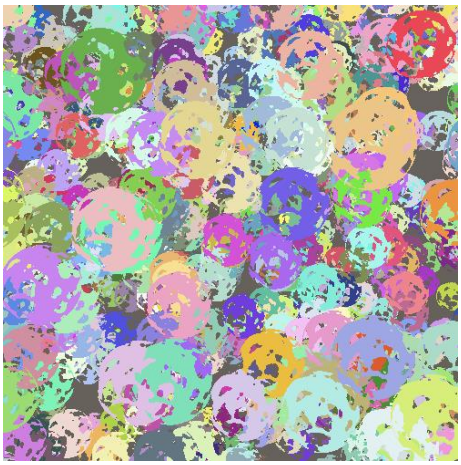
```
6.  [SerializeField]
7.  Color baseColor = Color.white;
8.
9.  [SerializeField, Range(0f, 1f)]
10. float cutoff = 0.5f;
11.
12. …
13.
14. void OnValidate () {
15.     …
16.     block.SetColor(baseColorId, baseColor);
17.     block.SetFloat(cutoffId, cutoff);
18.     GetComponent<Renderer>().SetPropertyBlock(block);
19. }
```



基于每个实例对象的 Alpha Cutoff

3.7. 透明裁剪的多球体球

对于 MeshBall 也是如此。现在我们使用了镂空材质，但所有的实例都显示出完全相同的洞。

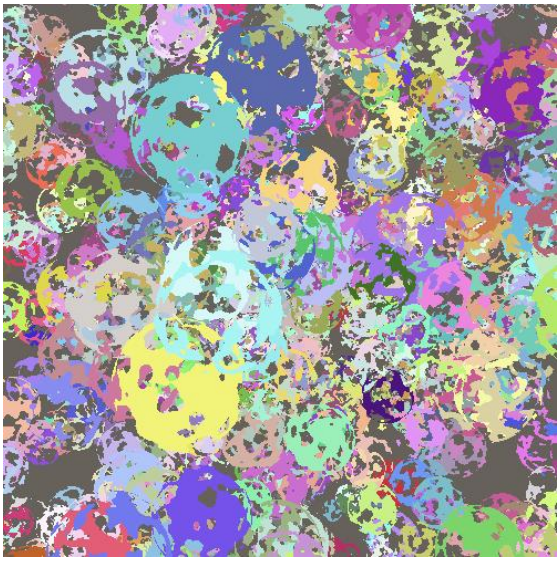


Alpha 裁剪网格球

让我们通过给每个实例一个随机旋转，再加上一个范围 0.5-1.5 的随机均匀缩放来添加一些

变化。但是预期设置每个实例的裁剪阈值，不如把他们颜色的 alpha 通道改成 0.5-1 的随机值。这使得我们的控制不太精确，但毕竟还是一个随机例子。

```
matrices[i] = Matrix4x4.TRS(
    Random.insideUnitSphere * 10f,
    Quaternion.Euler(
        Random.value * 360f, Random.value * 360f, Random.value * 360f
    ),
    Vector3.one * Random.Range(0.5f, 1.5f)
);
baseColors[i] =
    new Vector4(
        Random.value, Random.value, Random.value,
        Random.Range(0.5f, 1f)
    );
```



更多不同的网格球

需要注意的是，Unity 依然想 GPU 发送了一个 cutoff 值的数组，每个实例对应一个值，即使它们是完全相同的。这个值是材质属性的副本，所以改变它可以一次性改变所有球体的孔洞，尽管它们仍然是不同的。

这个不受光着色器目前为下个教程中将要创建的更复杂的着色器提供了一个很好的基础。

下一篇是平行光。