Unity SRP学习

一、Custom Render Pipeline（RP）

Unity 2018：the Lightweight RP and the High Definition RP.

Unity 2019：the Universal RP and the High Definition RP.

本教程第一阶段将实现一个绘制Unlit材质的自定义管线

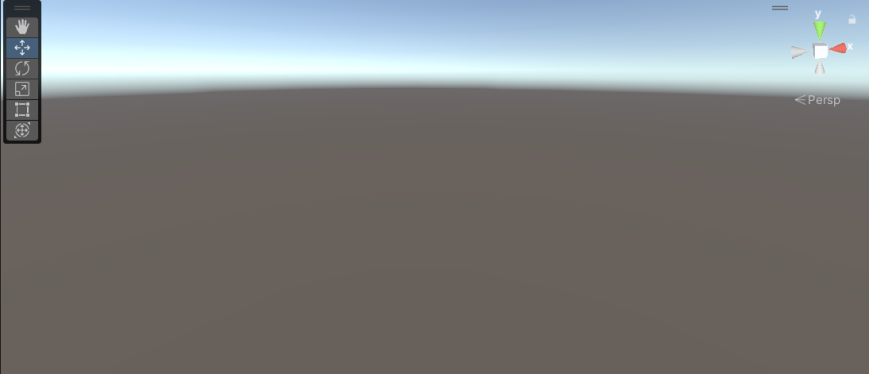
创建自定义的RP，首先需要一个CustomRenderPipelineAsset类和CustomRenderPipeline类。

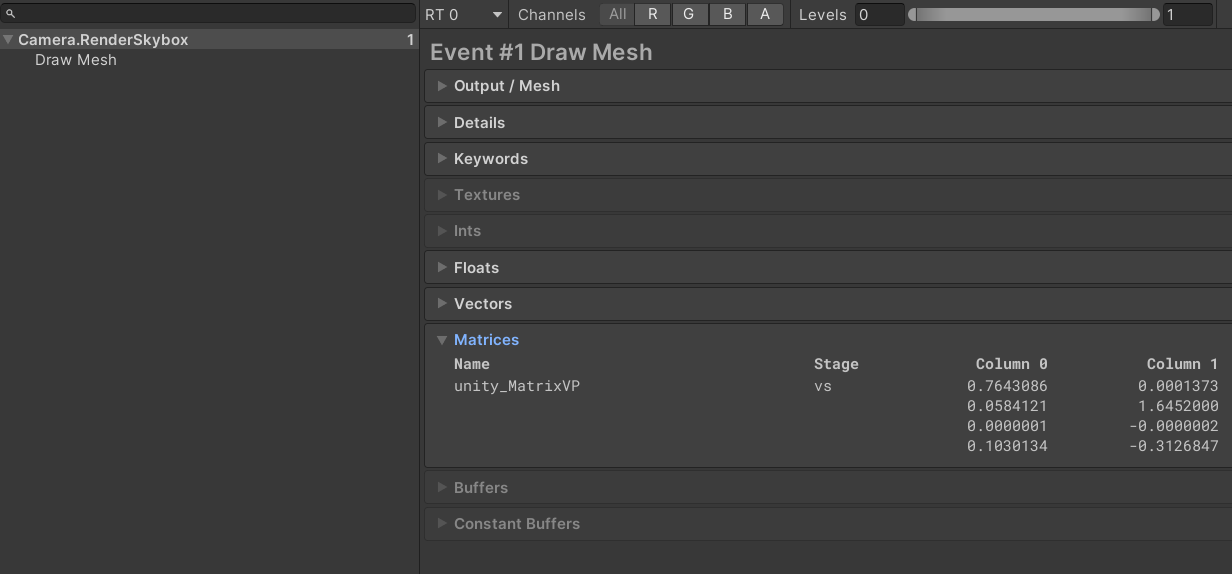
前者重写CreatePipeline方法，返回后者的一个实例，后者通过Render方法作为自定义SRP的入口。

注册自定义的CameraRenderer类，包含ScriptableRenderContext和Camera两个成员，在CustomRenderPipeline类的Render方法中，遍历所有相机并调用该Renderer类的绘制接口。

首先尝试绘制天空盒，分成三步：1.向context注册相机属性；2.调用DrawSkybox；3.提交context中的命令。

得到：





注：可以通过 Window / Analysis / Frame Debugger 打开帧调试器查看绘制步骤。

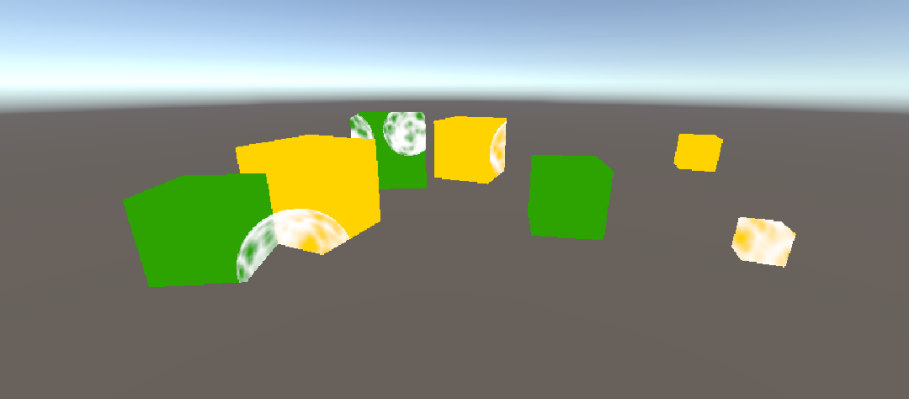
为了绘制其他物体，需要创建命令缓冲区，作为renderer类的一个成员变量。

在设置上下文时，清空缓冲区并开始采样；在提交上下文前，停止采样，并向上下文内写入命令缓冲区的命令。

为了获取需要绘制的物体，在每帧的开始，将相机的视锥体信息提交给context，获取视锥体剔除后的物体信息。

然后，在DrawVisibleGeometry中，通过相机设置物体排序方式（sortingSettings），通过shaderTagId和排序方式设置绘制方式（drawingSettings），通过指定RenderQueueRange设置过滤方式（filterSettings），再把它们和剔除结果一起传给context的DrawRenderer方法：

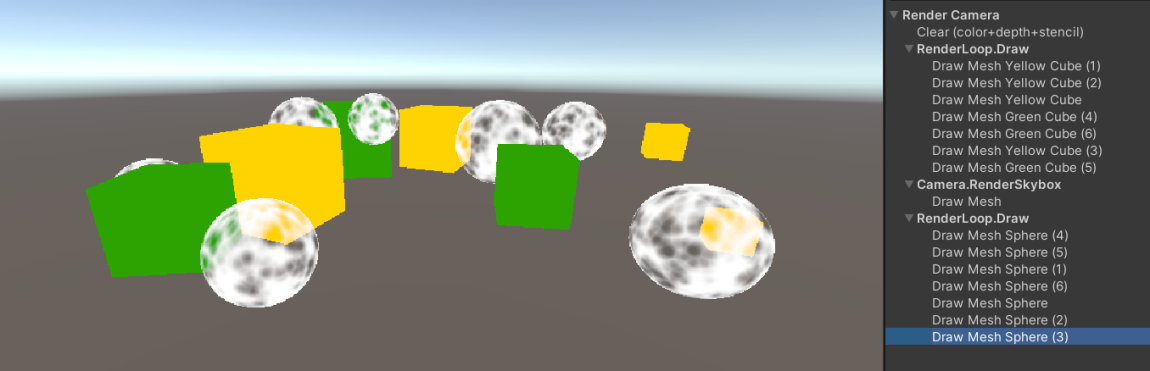
shaderTagId = SRPDefaultUnlit



因为先绘制透明物体，再绘制天空盒，且前者未写入深度缓冲区，所以显示结果如上所示。

因此，在第一次调用DrawRenderer时，限制过滤范围为不透明物体；在绘制天空盒后，再调用DrawRenderer，限制过滤范围为透明物体。

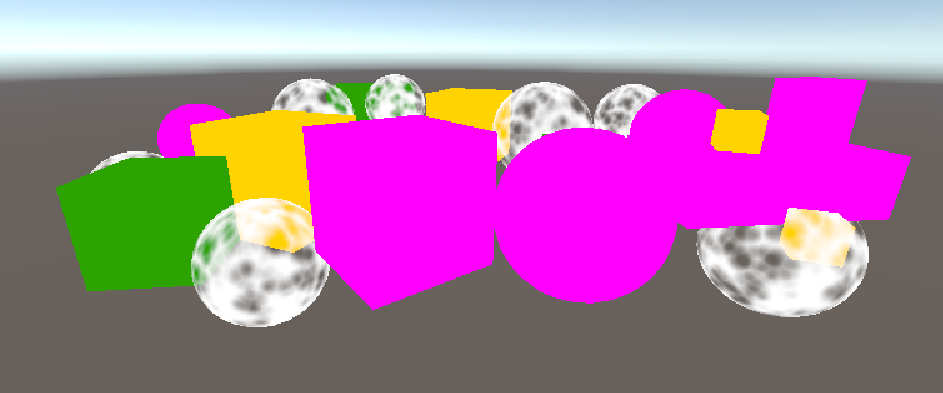
得到正确的结果：



（透明物体按实例从后向前绘制，但这并不是完全正确的）

下一步，准备拓展绘制的对象范围。

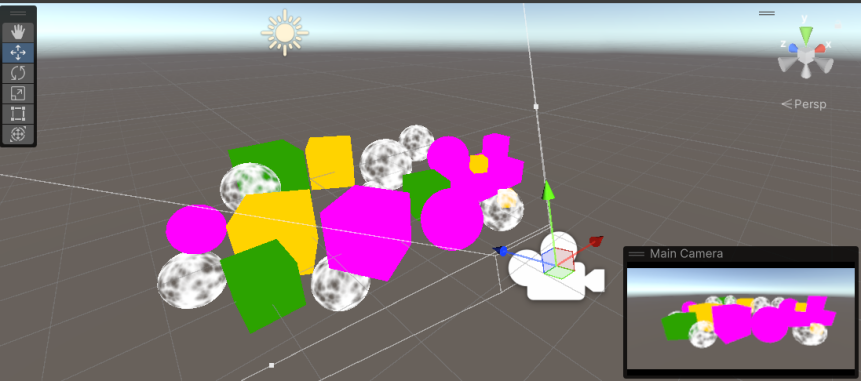
记录所有合法的ShaderTagId，依次注册给drawingSettings里的shaderPassName，再用“错误”材质替代原有材质，并使用默认的filterSetting绘制：



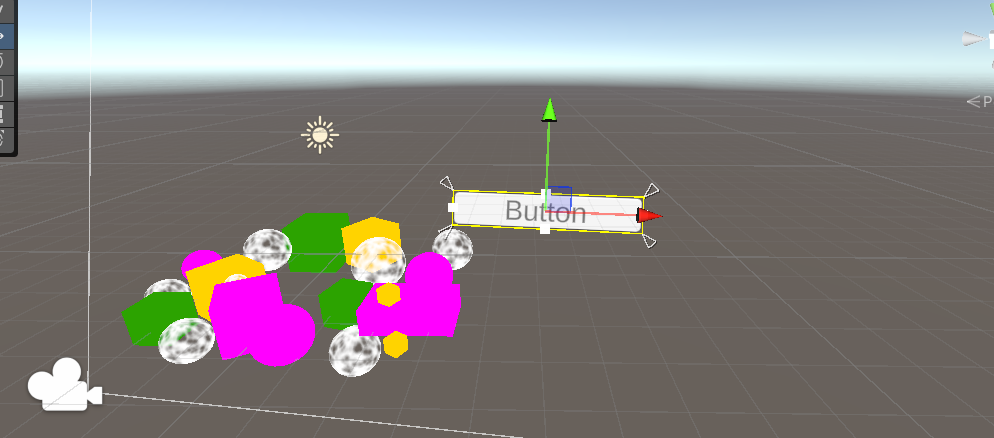
进一步的，通过partial关键字分离代码，并通过#if UNITY\_EDITOR控制“错误”材质的绘制。

下一步，准备绘制Gizmo。该步骤在所有绘制的最后执行。

通过Handles.ShouldRenderGizmos判断是否绘制Gizmo，然后直接调用context的对应接口即可，要对PreImage的部分和PostImage的部分都进行调用。



为了在编辑界面绘制游戏内UI，先把主相机传给Canvas，再在视锥剔除之前通过ScriptableRenderContext.EmitWorldGeometryForSceneView注册对应的几何体。



接下来，处理多个相机的情况，场景中存在多个相机时，会按照它们的"Depth"属性递增进行绘制。ClearFlags从1到4依次是 Skybox、Color、Depth 和 Nothing。因此在进行最开始的Clear时，可以根据相机的Flag调整设置。

调整第二个相机的ClearFlags和Viewport Rect属性，可以得到有趣的结果：



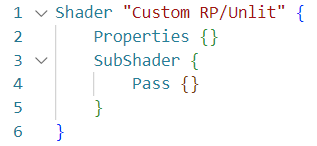
注：还可以通过Layer修改每个相机绘制的内容。

第一节结束。

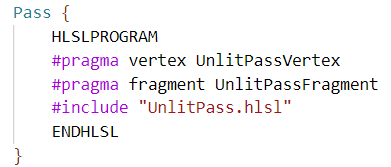
1. Draw Calls

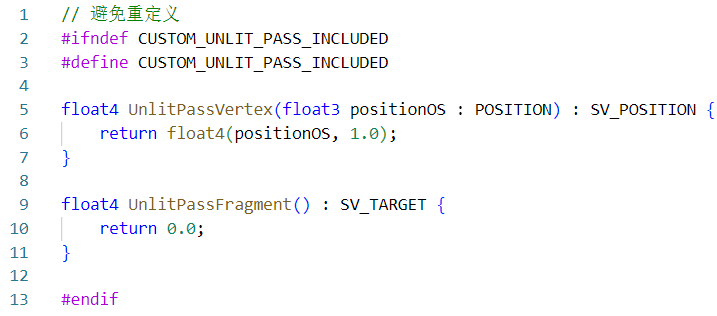
本部分包括编写着色器，以及高效绘制多个物体。

最基本的Shader结构：



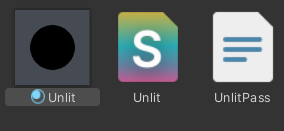
接下来，可以在Pass模块中写HLSL代码，我们将HLSL代码写到另一个文件再include进来，基本的结构是：



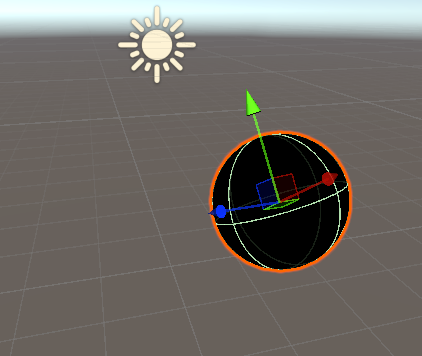


函数里最后冒号后跟的定义表示返回值的语义。

此时，用该着色器实例化出的材质已经可以显示效果了：



然后，通过MVP矩阵变换输入的顶点坐标，作为顶点着色器的输出。对应的uniform为：unity\_ObjectToWorld和unity\_MatrixVP。此时，可以在正确的位置绘制物体：

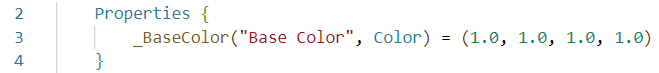


在“Core RP Library”包中，包含了更多的辅助函数，可以通过如下方式引用，使用该头文件的函数需要预定义一些宏，具体的可以查看项目中的代码。

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

（不过我觉得这样可能会导致不必要的代码膨胀，或许unity编译时会优化）

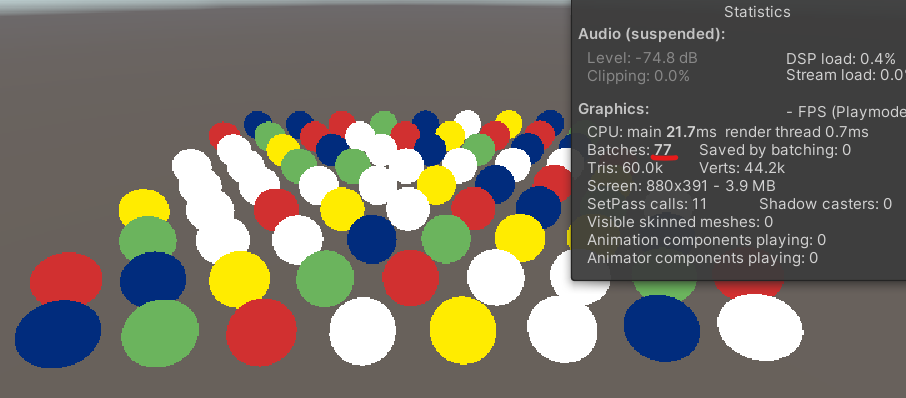
可以在Property块中定义输入的uniform变量，然后在着色器代码中使用：



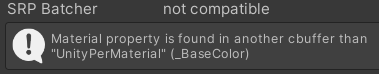


下面处理多个物体的draw call。

创建76个球体，共5种材质，当前有77个Draw Call：球体+天空盒+Clear（忽略）。

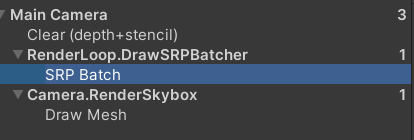


SRP Batcher不会减少Draw Call数量，而是会将材质属性缓存在 GPU 上，因此不必在每次绘制调用时都发送这些属性。但是，我们的Shader需要满足对应的要求。



因此，我们需要在具体的内存缓冲区（cbuffer）定义材质属性，具体见代码。

除了解决兼容问题外，还需要在构造Custom Render Pipeline时启用GraphicsSettings. useScriptableRenderPipelineBatching

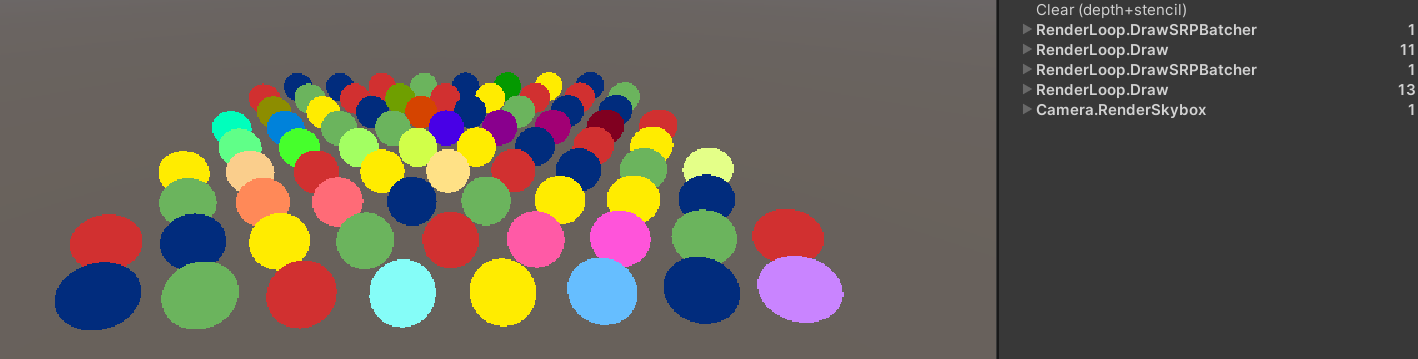


注意到，这里的合并指的是一个优化序列，而不是一个Draw Call。

因为SRP Batch要缓存数据，所以每个材质的内存布局必须相同，每个绘制调用只需要包含到正确内存位置的偏移量。

接下来，通过脚本为每个球体设置不同颜色。

在OnValidate里向MarterialPropertyBlock修改对应的uniform，再传给Renderer这个Component即可。向场景里创建24个不一样颜色但是同样材质的球体，是可行的，但是它们的SRP Batch优化被取消了。

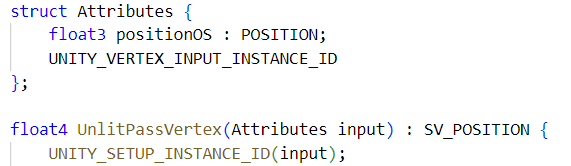


接下来，尝试使用GPU instancing。

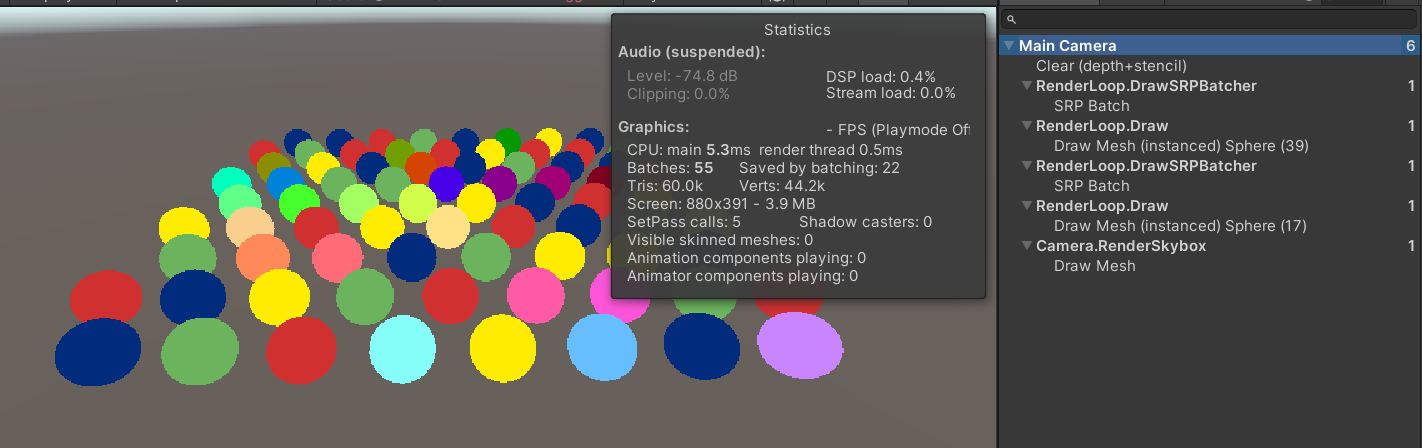
先启用instancing的着色器变体：#pragma multi\_compile\_instancing

接着，我们需要重新定义Matrix的宏，通过数组索引对应数值，通过core RP里的头文件覆盖自己的实现。除此之外，我们还需要知道当前处理的实例索引号。

然后，修改顶点着色器的输入：

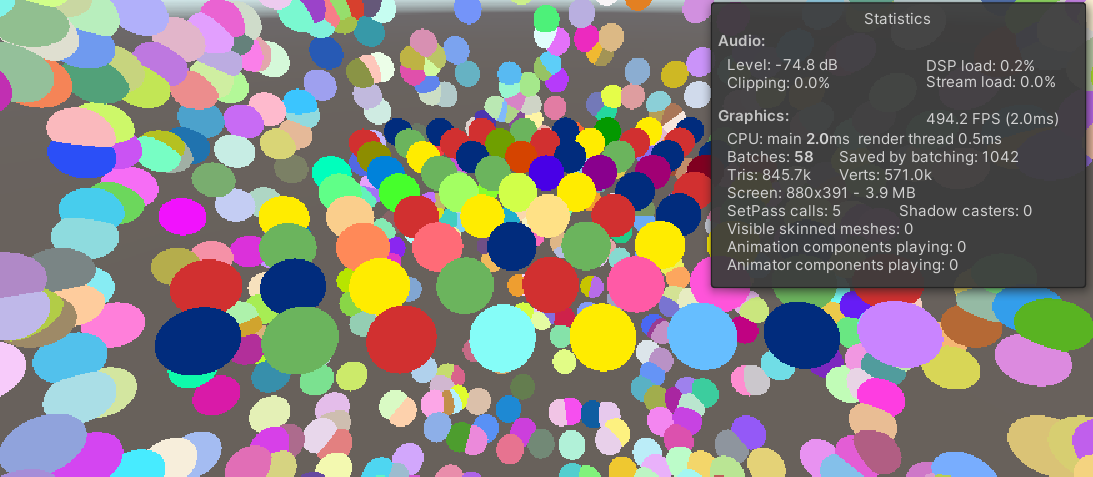


同理，我们也要修改顶点着色器的输出，以便在片元着色器中通过UNITY\_ACCESS\_ INSTANCED\_PROP访问具体的颜色。



可以看到，24个球体的Draw Call节省了22个。网站提示说如果每个实例的内存占用比较多，可能会导致Batch被分开。

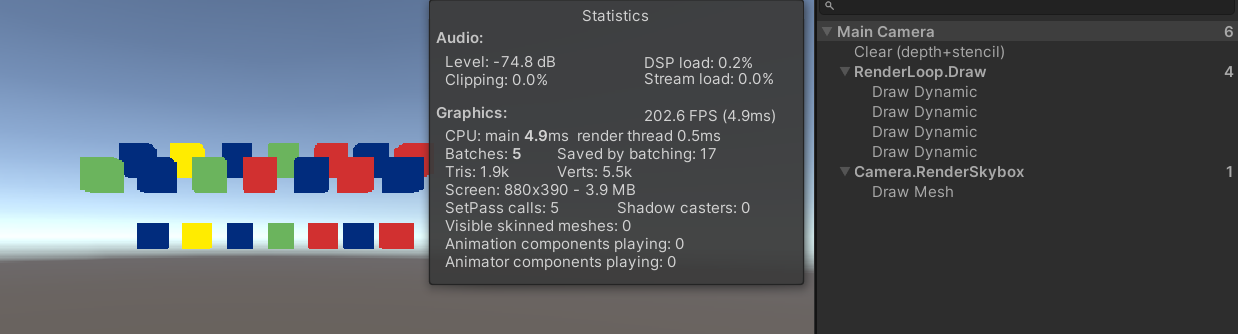
可以编写脚本生成1023个球体的随机位置和颜色，通过Graphics.DrawMeshInstanced实现实例绘制（代码见MeshBall）：



只多了3个Draw Call，这里的数量与单个Draw Call的缓冲区内存限制有关

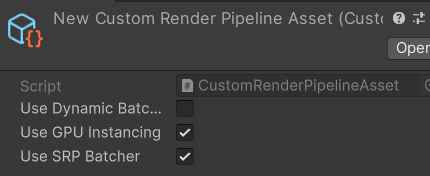
注意，网格体的绘制顺序与我们提供数据的顺序相同。

还有第三种减少Draw Call的方法，叫动态批处理，它可以把共享一个材质的多个小网格体合并为一个大网格体。可以在Pipeline中绘制Unlit的DrawSettings中打开，同时需要关闭Instancing，并把之前的SRP Batching关闭。



网格体要尽可能简单，不能使用每个物体设置不同材质属性的方法，合并的结果如上所示。并且，因为合并为单个网格，所以网站说，当涉及不同的尺度时，不能保证较大网格的法向量是单位长度，此外，绘制顺序也会发生变化。

此外，还有静态批处理，但那是预先合并网格的方法，Pipeline无需关注这点。



最后，把这三种优化的配置设置为Custom RP Asset的可配置内容，在CreatePipeline时传给流水线即可。