2 Draw Call

Set Pass Call代表改变渲染状态，当切换材质或者切换同一材质中Shader的不同Pass进行渲染时都会触发一次Set Pass Call。比如我们渲染1000个相同的物体和渲染1000个不同的物体，虽然两者Draw Call都是1000，但是前者Set Pass Call为1，后者还是1000。

切换渲染状态往往比Draw Call更耗时，所以这也是URP不再支持多Pass的原因。

如果Draw Call数量过多，CPU就会把大量时间花费在提交Draw Call上，造成CPU过载，游戏帧率变低，所以我们需要使用批处理（Batching）技术来降低Draw Call。

早期的Unity只支持动态批处理和静态批处理，后来又支持了GPU Instancing，最后SRP出现时支持了一种新的批处理方式SRP Batcher。

2.2.2 SRP Batcher

SRP Batcher是一种新的批处理方式，它不会减少Draw Call的数量，但可以减少Set Pass Call的数量，并减少绘制调用命令的开销。CPU不需要每帧都给GPU发送渲染数据，如果这些数据没有发生变化则会保存在GPU内存中，每个绘制调用仅需包含一个指向正确内存位置的偏移量。

SRP Batcher是否会被打断的判断依据是Shader变种，即使物体之间使用了不同的材质，但是使用的Shader变种相同就不会被打断，传统的批处理方式是要求使用同一材质为前提的。

SRP Batcher会在主存中将模型的坐标信息、材质信息、主光源阴影参数和非主光源阴影参数分别保存到不同的CBUFFER（常量缓冲区）中，只有CBUFFER发生变化才会重新提交到GPU并保存。

1. 它是指材质的所有属性都需要在常量内存缓冲区CBUFFER里定义，要我们将\_BaseColor这个属性在名字为UnityPerMaterial的CBUFFER块中定义，但并非所有平台（如OpenGL ES 2.0）都支持常量缓冲区，我们使用SRP源码库中的CBUFFER\_START和CBUFFER\_END宏来替代CBUFFER块。这样的话不支持常量缓冲区的平台就会忽略掉CBUFFER的代码。

2. 我们在UnlitPass.hlsl中将\_BaseColor定义在名字为UnityPerMaterial的常量缓冲区中。

3. 在UnityInput.hlsl中把几个矩阵定义在UnityPerDraw的常量缓冲区中。

4. 它指出，如果我们需要使用一组特定值的其中一个值，我们需要把这组特定值全部定义出来，现在还缺少unity\_LODFade的定义。

5. 至此，我们的Shader已经兼容SRP Batcher了，我们在代码中启用SRP Batcher进行测试。创建渲染管线实例的时候，在构造函数里启用。

若想给相同的物体设置不同的颜色，那么每个物体都需要使用一个不同的材质并调整颜色，我们接下来编写一个脚本，让所有相同物体使用同一个材质，但可以给每个物体设置不同的颜色。

在CustomRP下创建一个Examples子文件夹，新建脚本PerObjectMaterialProperties.cs，脚本中我们定义一个可以调整颜色的baseColor属性，并将颜色值通过MaterialPropertyBlock对象传递给材质。把这个脚本挂到每一个球体上面，然后设置不同的颜色。

但我们发现SRP Batcher失效了，没有办法处理每个对象的材质属性。

​2.2.4 GPU Instancing

如果能将数据一次性发送给GPU，然后使用一个绘制函数让渲染流水线利用这些数据绘制多个相同的物体将会大大提升性能。这种技术就是GPU多例化（GPU Instancing）技术。使用GPU Instancing能够在一个绘制调用中渲染多个具有相同网格的物体，CPU收集每个物体的材质属性和变换，放入数组发送到GPU，GPU遍历数组按顺序进行渲染。

假设需要渲染100个相同的模型，每个模型有256个三角形，那么需要两个缓冲区，一个是用来描述模型的顶点信息，因为待渲染的模型是相同的，所以这个缓冲区只存储了256个三角形（如果不存在任何的优化组织方式，则有768个顶点）；另一个就是用来描述模型在世界坐标下的位置信息。例如不考虑旋转和缩放，100个模型即占用100个float3类型的存储空间。

1. 要支持GPU Instancing，首先需要在Shader的Pass中添加#pragma multi\_compile\_instancing指令，然后在材质球上就能看到切换开关了，这时Unity会为我们的Shader生成两种变体。

2. 在Common.hlsl文件中include SpaceTransforms.hlsl之前，我们将SRP源码库中的UnityInstancing.hlsl文件Include进来，我们需要用到里面的一些定义好的宏和方法。

3. UnityInstancing.hlsl通过重新定义一些宏去访问实例的数据数组，它需要知道当前渲染对象的索引，该索引是通过顶点数据提供的。UnityInstancing.hlsl中定义了UNITY\_VERTEX\_INPUT\_INSTANCE\_ID宏来简化了这个过程，但它需要我们存在一个顶点输入结构体，我们定义它并将positionOS的定义放进来，然后在结构体中加入UNITY\_VERTEX\_INPUT\_INSTANCE\_ID宏，最后该结构体对象作为顶点函数的输入参数。然后在顶点函数添加UNITY\_SETUP\_INSTANCE\_ID(input)代码，用来提取顶点输入结构体中的渲染对象的索引，并将其存储到其他实例宏所依赖的全局静态变量中。

4. 目前我们还不支持每个物体实例的材质数据，且SRP Batcher优先级比较高，我们还不能得到想要的结果。首先我们需要使用一个数组引用替换\_BaseColor，并使用UNITY\_INSTANCING\_BUFFER\_START和UNITY\_INSTANCING\_BUFFER\_END替换CBUFFER\_START和CBUFFER\_END。

5. 我们还需要在片元函数中也提供对象的索引，通过在顶点函数中使用UNITY\_TRANSFER\_INSTANCE\_ID(input，output)将对象位置和索引输出，若索引存在则进行复制。为此我们还需定义一个片元函数输入结构体，在其中定义positionCS和UNITY\_VERTEX\_INPUT\_INSTANCE\_ID宏。

6. 在片元函数中也定义UNITY\_SETUP\_INSTANCE\_ID(input)提供对象索引，且现在需要通过UNITY\_ACCESS\_INSTANCED\_PROP(UnityPerMaterial, \_BaseColor)来访问获取材质的颜色属性了。

​2.2.5 绘制许多网格小球

我们无需生成多个对象，只需要填充变换矩阵和颜色的数组，告诉GPU用它们去渲染Mesh，这样最多可以一次提供1023个实例，这是GPU Instancing的特性。然后我们在Awake方法中随机生成位置和颜色填充数组。最后调用Graphics.DrawMeshInstanced绘制网格。

2.2.6 动态合批

动态批处理的原理是每一帧把可以进行批处理的模型网格进行合并，再把合并好的数据传递给CPU，然后使用同一个材质进行渲染。好处是经过批处理的物体仍然可以移动，这是由于Unity每帧都会重新合并一次网格。

动态批处理有很多限制，比如在使用逐对象的材质属性时会失效，网格顶点属性规模要小于900等等，该技术适用于共享材质的小型的网格。

2.3 Alpha Blend和Alpha Test

在Unity中我们通常使用两种方法来实现透明效果：第一种是透明度测试（Alpha Test），这种方法其实完全无法得到真正的半透明效果；另一种是透明度混合（Alpha Blend）。

当使用透明度混合时，我们会关闭深度写入（ZWrite）。

透明度测试（Alpha Test）和透明度混合（Alpha Blend）的基本原理如下：

（1）透明度测试：采用极端霸道的方式，只要一个片元的透明度不满足条件（通常是小于某个阈值），那么它对应的片元就会被舍弃。被舍弃的片元将不再进行任何处理，也不会对颜色缓冲产生影响，否则就按照普通不透明物体的处理方式来处理它，即进行深度测试，深度写入等。透明度测试是不需要关闭深度写入的，它和其它不透明物体最大的不同就是它会根据透明度来舍弃一些片元。虽然简单但产生的效果也很极端，要么完全透明要么完全不透明，而且它会使得硬件底层的优化技术Early-Z失效（将深度测试提前到片元着色器之前，减少片元计算量，减少overdraw）。

（2）透明度混合：这种方法可以得到真正的半透明效果，它会使用当前片元的透明度作为混合因子，与已经存储在颜色缓冲中的颜色值进行混合，得到新的颜色。但是透明度混合需要关闭深度写入，这使得我们要非常小心物体的渲染顺序。需要注意的是，透明度混合只是关闭了深度写入，但没有关闭深度测试。这意味着当使用透明度混合渲染一个片元时，还是会比较它的深度值与当前深度缓冲的深度值，如果它的深度值距离摄像机更远，那么就不会再进行混合操作。这一点决定了，当一个不透明物体出现在一个透明物体的前面，即使我们先渲染了不透明物体，它仍然可以正常地遮挡住透明物体，对于透明度混合来说，深度缓冲是只读的。

2.3.2 材质添加对纹理的支持

1. 我们的材质球目前还不支持使用纹理，现在添加这个功能，在属性栏声明一张纹理。

2. Unity会自动将使用的纹理上传到GPU内存中，然后使用TEXTURE2D()宏定义一张2D纹理，并使用SAMPLER（sampler+纹理名）这个宏为该纹理指定一个采样器。纹理和采样器是着色器资源，必须在全局定义，不能放入缓冲区中。除此之外还需要获取纹理的平铺和偏移值，这是通过定义一个float4类型的纹理名\_ST属性来获取的，该属性可以在UnityPerMaterial缓冲区中定义，设置给每个对象实例。

3. 要采样纹理，我们还需要一套UV坐标，它应该被定义在顶点输入结构体中，纹理坐标要传到片元函数中进行采样，所以片元输入结构体也要定义UV坐标。

4. 在顶点函数中，传递纹理坐标之前把为纹理的缩放和偏移也计算在内。

5. 最后我们将片元函数通过SAMPLE\_TEXTURE2D宏对纹理采样，采样结果和颜色值相乘得到最终表面颜色。

2.3.3 透明度测试（Alpha Test）

2. 在UnityPerMaterial缓冲区中定义该属性，随后在片元函数中使用clip()函数舍弃不满足阈值的片元，它会判断传参如果为负数，就会舍弃当前像素的输出颜色（该片元就会产生完全透明的效果）。

材质通常使用透明度测试和透明度混合其中一个，而不是同时使用。透明度测试应使用在完全不透明的物体身上，除了被clip丢弃的片元外，其它片元会写入深度缓冲中。我们把混合模式设置成标准不透明物体的配置，然后开启深度写入，渲染队列设置为AlphaTest。

2.3.4 Shader Feature

使用shader feature可以让Unity根据不同的定义条件或关键字编译多次，生成多个着色器变体。然后通过外部代码或者材质面板上的开关来启用某个关键字，加载对应的着色器变种版本来执行某些特定功能，是项目开发中比较常用的一种手段。下面我们的目标是添加一个控制透明度测试功能是否启用的开关。

1. 首先添加一个控制着色器关键字的Toggle切换开关来控制是否启用透明度测试功能。

2. 在Pass中使用shader feature声明一个Toggle开关对应的\_CLIPPING关键字。

3. 然后在片元函数中通过判断该关键字是否被定义，来控制是否进行裁剪操作。

接下来就可以在材质面板中将控制裁剪功能启用。