**2 Draw Call**

**2.1 Shader**

三种着色语言：微软DirectX的HLSL、OpenGL的GLSL和NVIDIA的CG。

CG已经停止更新多年，基本上已经被放弃了。现在SRP的着色器代码库使用的是HLSL，Unity也使用了HLSL的编译器来编译Shader，且HLSL转GLSL比较容易。

**2.1.1 Unlit Shader**

为了避免重复声明，使用宏定义判断否是已经声明。

**2.1.2 着色器函数**

**2.1.3 SRP源码库**

**2.2 批处理**

**2.2.1 Draw Call和Set Pass Call**

Set Pass Call表示改变渲染状态，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。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。

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支持SRP Batcher。

**2.2.2 SRP Batcher**

SRP Batcher不会减少Draw Call的数量，但可以减少Set Pass Call的数量，并减少Draw Call命令的开销。

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. 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。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. 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。

**2.2.3 多种颜色**

我们编写一个脚本，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。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。

对应的顶点着色器和片元着色器也要做好对应的设置才能使用多例化技术。

1. 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. 在顶点输入结构体使用**UNITY\_VERTEX\_INPUT\_INSTANCE\_ID**宏获取当前渲染对象的索引。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. 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. 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，在片元函数中也提供对象的索引。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**

我们使用脚本MeshBall.cs来生成多个Mesh和多个小球对象，来展示成百上千个对象使用GPU Instancing进行合批的效果。

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。

我们的渲染管线已经支持了三种批处理，将这些批处理的启用开关设置成可配置项，使用或禁用哪种批处理由用户指定，在CameraRenderer.DrawVisibleGeometry()方法中作为参数传入。

**2.3 Alpha Blend和Alpha Test**

Unity中有两种方法实现透明效果：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。

透明度测试会使Early-Z失效。

**2.3.1 Blend Modes**

**2.3.2 材质添加对纹理的支持**

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。

**2.3.3 透明度测试**

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。

**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。

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

**2.3.5 逐对象裁剪**

我们在PerObjectMaterialProperties.cs脚本中也添加裁剪的属性，可以给每个对象设置不同的裁剪程度，和设置颜色属性时差不多。