# Unity渲染优化

在Unity中的渲染中一个新的材质使用的时候，需要准备进行渲染设置工作（SetPass），这部分耗时是一个Draw Call的主要耗时点，所以如果场景有越多的materials，就会有越多的CPU必须使用去设置GPU 数据。传统的优化做法是减少Draw Call数量去提升CPU渲染性能，而实际上真正的CPU消耗来自那些设置工作，而不是GPU Draw Call本身；

首先在Unity渲染前Unity会去收集我们场景中所有可见的渲染信息，比如我们场景物体的的网格，材质，纹理等；

然后根据相机的视图和剔除信息，决定哪些对象需要被渲染，将需要渲染的物体放到渲染队列里，按照材质和透明度进行分组，来优化渲染顺序，CPU收集和计算将要被渲染的物体的相关信息并发送命令给GPU（Draw Call），CPU 会给每一个 Draw Call 创建数据包Batch（批次），每一个批次至少包含一个 Draw Call，CPU会去处理批次；

遍历每个可渲染对象，Unity会根据对象的材质和着色器去决定如何渲染，在这个过程里会涉及到多个Pass和SetPass的调用；

Pass是我们Shader用于描述渲染过程的一个部分，他定义了物体的顶点和片段数据，用来实现不同的渲染效果；

当Unity准备开始绘制一个对象时，会调用材质的SetPass方法来设置当前的渲染状态；

每个材质的Pass都会调用一次SetPass，如果一个材质使用了多个Pass，Unity会依次调用每个Pass的SetPass，来准备渲染该物体；然后每一次SetPass都会调用一次Draw Call，这就会造成性能开销；

这时候就用到了我们的优化技术，比如说合批，GPU Instancing，SRP Batcher，合并纹理图集等；

合批又分为静态合批和动态合批；

静态合批是将多个静态对象的几何数据合并成一个大的几何数据，这样Unity在渲染这些对象的时候只需要一次SetPass和一次Draw Call就可以了，使用静态合批需要我们开发人员将需要静态合批的对象标记为静态（static）这意味着这些对象在游戏运行中不会发生移动、缩放、旋转等，完全是一个静止的状态；非常适合用于大规模的静态场景；但是他会增加场景的构建时间和内存使用；

动态合批是指在运行时，Unity自动将符合合批条件的小型动态对象合并到一个大的顶点缓冲区中（不需要去勾选静态标记），合并后作为一个单独的批次进行渲染；动态合批的限制不光需要对象使用相同的材质，对顶点也有一定的限制，通常限制在900个顶点和300个顶点索引以内，如果超过这个限制就不会进行动态合批；然后每次更新都会去重新计算合批数据，所以频繁的更新对象也会造成一定的性能开销；

合批的好处在于，当又多个对象使用了相同的材质（材质的Pass一致）时，Unity可以通过合批技术合并这些对象，从而减少对SetPass的调用次数。

比如说：我们的场景里包含了100个使用相同材质且有相同Pass的对象，通过我们的合批技术，Unity只需要去调用一次SetPass，然后通过一次Draw Call完成所有对象的绘制，

但是，如果着色器包含了多个Pass，即使通过合批，Unity任然需要为每个Pass去调用SetPass，比如一个着色器有两个Pass，合批的效果只限于减少每个Pass下的对象切换，而无法避免每个Pass都需要单独调用SetPass；

举个例子：

如果场景中有100 个使用相同Shader的对象，假设这个Shader有两个不同的Pass，但是没有合批。

首先Unity会为第一个Pass调用100次SetPass和100次Draw Cal。然后为第二个Pass再次调用100次SetPass和100次Draw Call。这样带来的性能消耗无疑是巨大的；

但是如果我们使用了合批在调用第一个Pass的时候我们只需要调用一次SetPass和Draw Call绘制这100个对象，同样在调用第二个Pass的时候也只需要调用一次SetPass和Draw Call 就可以了。通过合批，Unity将原来需要调用200次SetPass和200次Draw Call来绘制这100个对象优化到了仅需要2次SetPass和Draw Call，这无疑减少了大量的性能开销；

尽管说合批可以有效的减少SetPass调用的次数，但是合批是有局限性的；

比如：合批的对象必须使用相同的材质，才可以达到合批的条件，如果对象使用了不同的材质或者不同的Pass，那么合批将无法生效，任然会多次的去调用SetPass；如果说我们有一个材质有多个Pass，那么即使进行了合批，Unity也必须分别去处理每个Pass，从而多次的调用SetPass；

GPU Instancing 是通过共享相同的集合数据和材质属性来减少GPU渲染调用的过程，他会将多个相同的物体合并成一组，使用相同的材质。然后将这些实例化物体的变换矩阵等数据上传到GPU然后设置材质和其他的渲染状态，通过调用DrawMeshInstanced一次性渲染所有的实例，而不是依次调用Draw Call通过这种方式，SetPass 和 DrawCall 的数量显著减少，尤其是在需要渲染大量相同物体时，比如大量树木，草从，敌人等；但是因为需要储存多个实例的变换矩阵和额外的属性数据，会增加GPU内存的使用；

SRP Batcher 是Unity用于SRP渲染管线，LWRP轻量级渲染管线和HDRP高清渲染管线中的一个特性。他也是将多个网格合并成单个批次进行渲染与其他合批不同,SRP Batcher将未改变属性的Mesh缓存起来,从而减少消耗，SRP Batcher会将相同Shader的所有属性（如材质属性、材质实例）打包到一个大的数据缓冲区中，然后批量发送到GPU。GPU在执行这些批次时无需频繁切换Shader或重新加载属性，从而提高渲染效率。

但是SRP Batcher仅对基于SRP的Shader生效。Unity标准的内置渲染管线不支持SRP Batcher。

纹理图集合并

纹理图集合并就是，将许多小的纹理合并成一个大的纹理。这样可以在减少渲染过程中的SetPass调用和Draw Call，从而提高渲染性能。

通常情况下，每个材质使用一个纹理，当你渲染多个使用不同纹理的对象时，GPU必须频繁的去切换纹理，这无疑会产生许多性能上的开销，我们通过将多个小纹理合并为一个大的纹理，就可以通过一次SetPass来绘制多个使用不同小纹理的对象，避免频繁的材质切换和纹理绑定；

虽然合并纹理可以有效的减少材质的切换，但是大纹理的采样还是会增加GPU的负担，首先，当GPU从大纹理中采样时，可能会访问较大的纹理块，这会增加纹理缓存的压力，尤其是在大纹理跨越多个片段时，GPU需要频繁的从显存读取数据；如果说大纹理的纹理采样器使用的是双线性、三线性或各向异性过滤、采样大纹理时可能会涉及更多的纹理区域，这也会导致GPU进行更多的插值运算，增加负担；

然而，这种采样负担的增加通常比频繁的 SetPass 和 DrawCall 切换带来的性能消耗要小，所以在大多数情况下，合并纹理的优化效果还是可取的；

在我们渲染每个使用小纹理的对象时，整张的大纹理都会作为材质的输入被传给GPU，不过GPU只会根据对象的UV坐标在大纹理中采样对应的小区域，而不会去处理整张大纹理，所以我们应该把常用的纹理和不常用的纹理进行分开打包；

然后通常最大纹理尺寸的限制是4096xp\*4096xp或者8192xp\*8192xp；