Unity shader 渲染进界：细节或框架理解升华

deepwaterooo

September 29, 2018

Contents

1 Unity Shader 入门精要笔记（一）：渲染流水线

- https://blog.csdn.net/lzhq1982/article/details/73312811

1.1 一个渲染流程分为 3 个阶段

- 应用阶段（Application Stage）：cpu 负责，由开发者准备好摄像机，模型，光源等数据，做好粗粒度的剔除工作，并设置好材质，纹理，shader 等渲染状态，最后输出渲染所需要的图元的过程。
- 几何阶段（Geometry Stage）：gpu 负责，把上个阶段传过来的图元进行逐顶点，逐多边形的操作。把顶点坐标从模型空间经过一系列转换最终到屏幕空间，输出屏幕空间二维顶点坐标，顶点深度、着色等数据，交给光栅器处理。
- 光栅化阶段（Rasterizer Stage）：gpu 负责，对上个阶段的逐顶点数据（纹理坐标、顶点颜色等）进行插值，再逐像素处理，决定哪些像素应该被绘制在屏幕上，并计算他们的颜色。

1.2 CPU 和 GPU 的通信

1.2.1 cpu 的工作：把数据加载到显存，设置渲染状态，调用 draw call。

- 把数据加载到缓存：所有数据都要从硬盘加载到系统内存，然后网格和纹理，顶点位置、颜色，法线，纹理坐标等数据加载到显存中。
- 设置渲染状态：使用哪个顶点着色器和片元着色器，光源属性，材质等来渲染的过程，渲染状态不改，所有网格都会使用一种渲染状态。
- 调用 draw call：翻译过来就是绘制命令，由 cpu 发出，指向一个要被渲染的图元列表，通知 gpu 可以按照上面的设置渲染这些图元了。我们常说优化 draw call，是因为 cpu 准备上述数据是个复杂的过程，而 gpu 绘制却很快，如果 draw call 很频繁，cpu 光顾着准备数据了，gpu 又空等着，结果可想而知，所以每次发出 draw call，我们应该尽可能多准备数据给 gpu，这样就可以用尽量少的 draw call 达到通知 gpu 快速渲染的效果。可是如何调一次 draw call 尽可能多准备数据给 gpu 呢，这需要一个专门的主题了。

1.2.2 GPU 流水线

- 上图中绿色表示该流水线是完全可编程控制的，黄色表示可以配置但不可以编程的，蓝色表示 GPU 固定实现的，开发者没有控制权。虚线框表示该 shader 是可选的。
- 下面我们逐一看一下：

1. 顶点着色器
    - 顶点着色器：写 shader 人员必须要面对的重点部分。顾名思义，该阶段处理的是顶点数据，重点是这里是单个顶点数据，你别想访问其他顶点数据。我们要在这里完成顶点的坐标变换，把顶点从模型空间转换到齐次裁剪空间，再经透视除法，最终得到归一化设备坐标（NDC）。还有逐顶点光照，并准备好后面阶段需要的其他数据，比如纹理坐标，顶点颜色等。

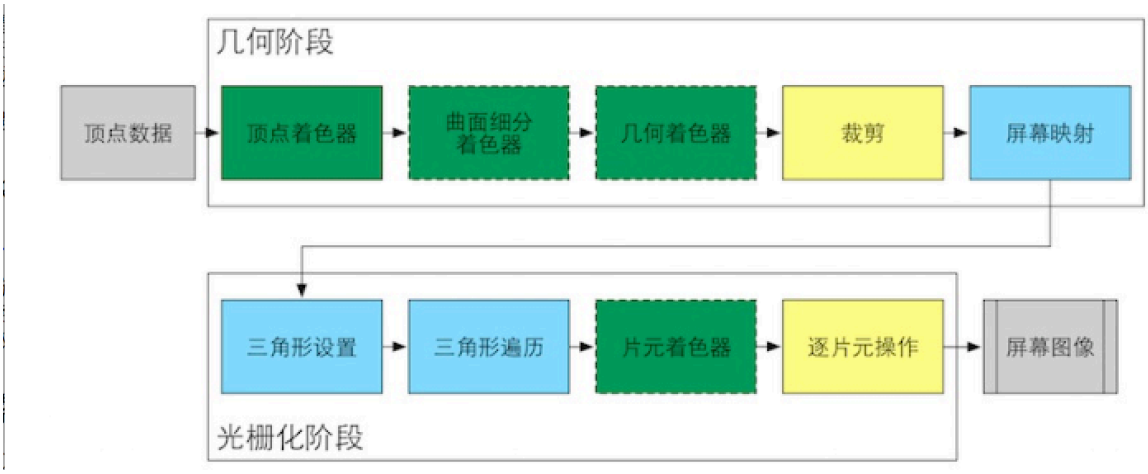
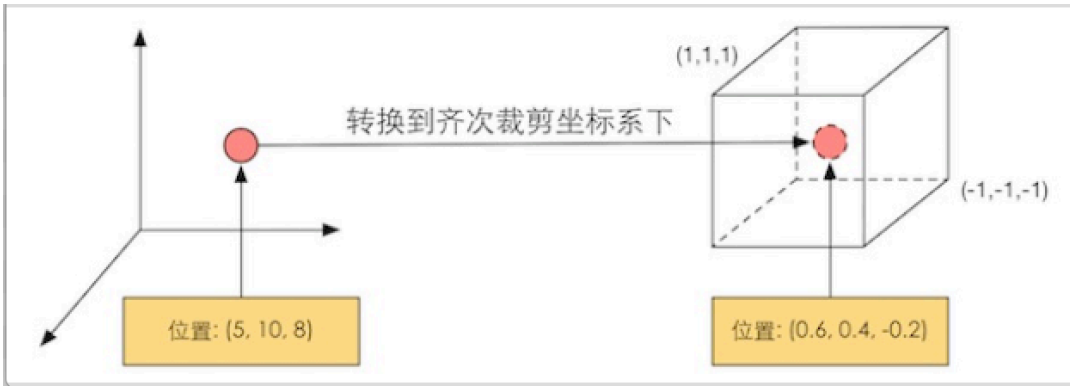


Figure 1: GPU 流水线

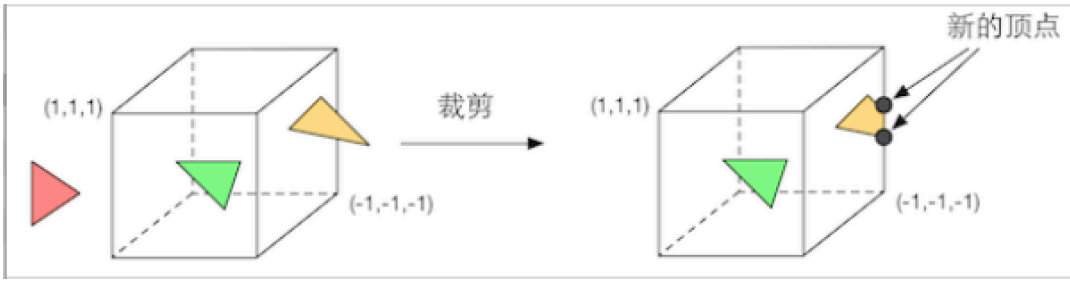


2. 曲面细分着色器

   - 曲面细分着色器: 用于细分图元，几何着色器用于执行逐图元的着色操作，或产生更多图元，这两个阶段都是可选的。

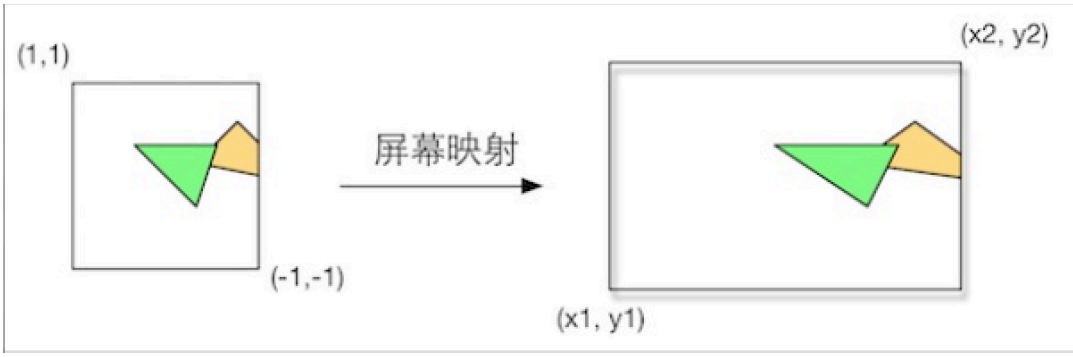
3. 裁剪

   - 裁剪：上面阶段我们得到的 NDC 是立方体空间，Unity，OpenGL 是从（-1,-1,-1）到（1,1,1）,DirectX 的 z 是从 0 到 1，完全在该空间内的图元保留，完全在外面的抛弃，一半在内一半在外的就需要裁剪了，这就是我们视野内的部分。该阶段是可配置的，比如我们用自定义的裁剪平面，或裁剪图元的正面或背面。



4. 屏幕映射

   - 屏幕映射：该阶段是不可配置和编程的。这里输入的坐标是三维的，屏幕映射的任务是把每个图元的 x，y 坐标转换到屏幕坐标系（二维）下，前面 NDC 我们的 x，y 在（-1，1）内，所以映射到屏幕上就是个缩放的过程，而最终的屏幕坐标系会和 z 坐标构成窗口坐标系，传给光栅化阶段。有一点要注意，OpenGL 会以屏幕左下角为最小坐标，而 DirectX 会以左上角为最小坐标。

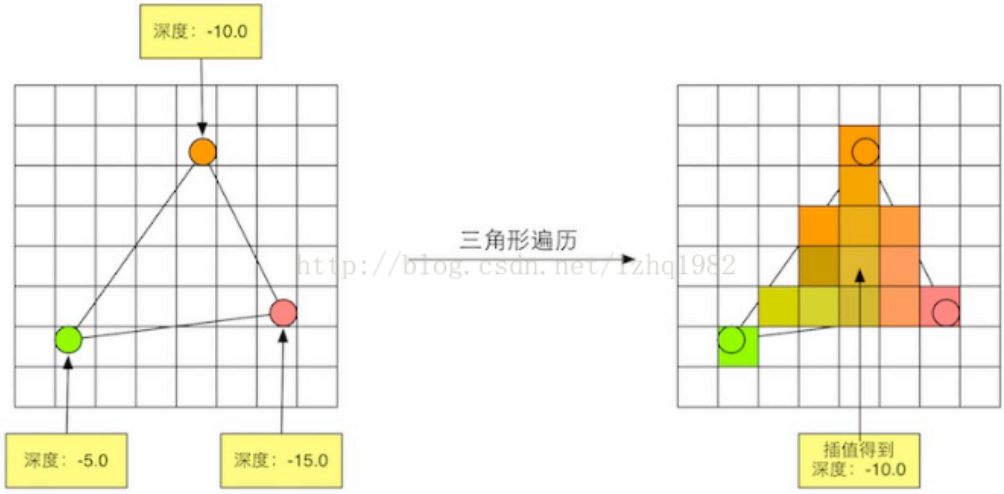


5. 三角形设置

- 三角形设置：不可配置和编程。从这一步开始进入光栅化阶段。上个阶段输出的都是三角形顶点，而这个阶段我们要用这些顶点和计算出来的三角形边界组成三角形网格数据。

6. 三角形遍历

- 三角形遍历：不可配置和编程。检查每个像素被哪个三角形所覆盖，而后生成片元，重点是三角形的三个顶点会对覆盖的每个像素的数据进行插值，比如深度和坐标等，记住这点很重要，如下图所示：

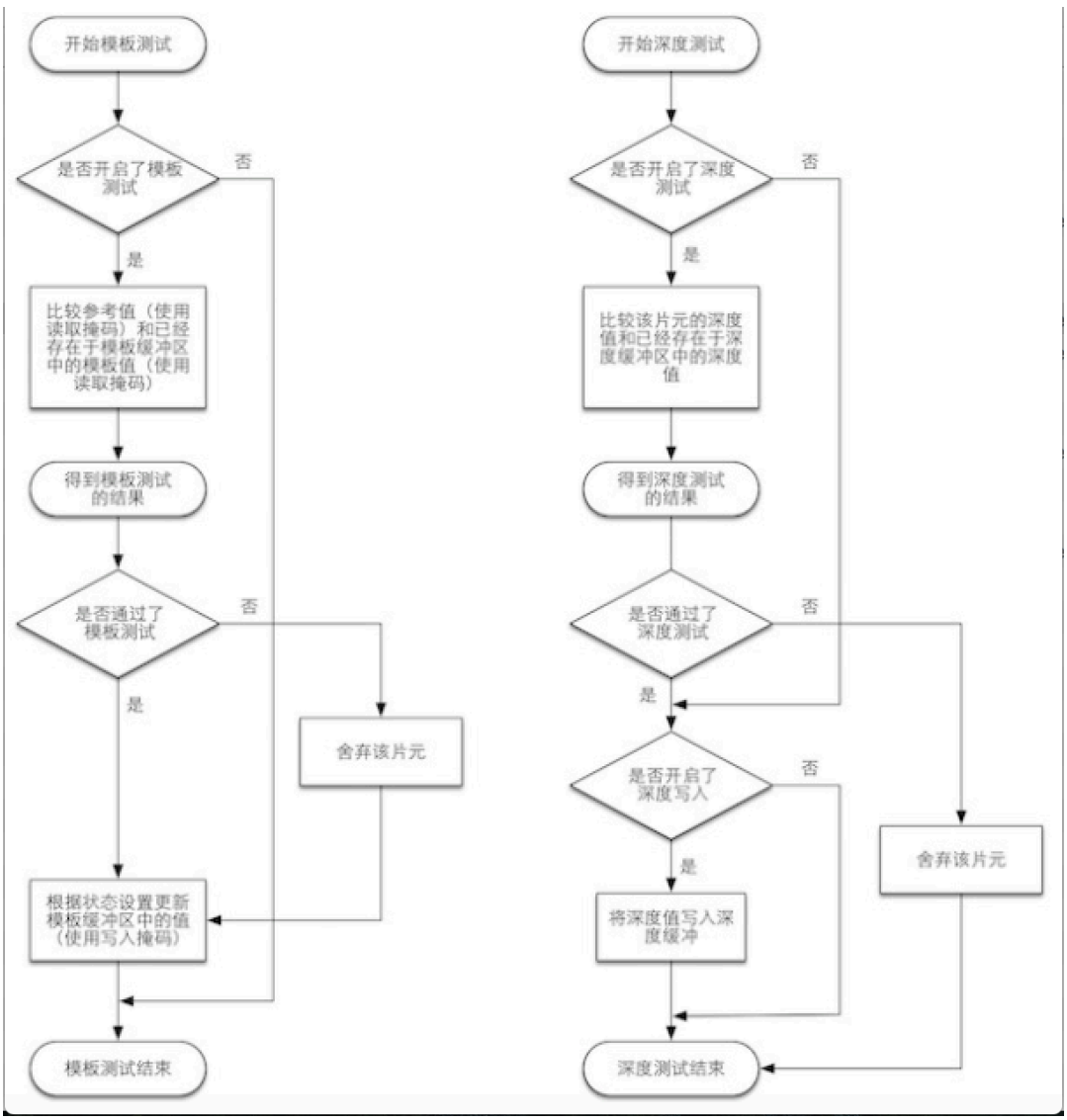


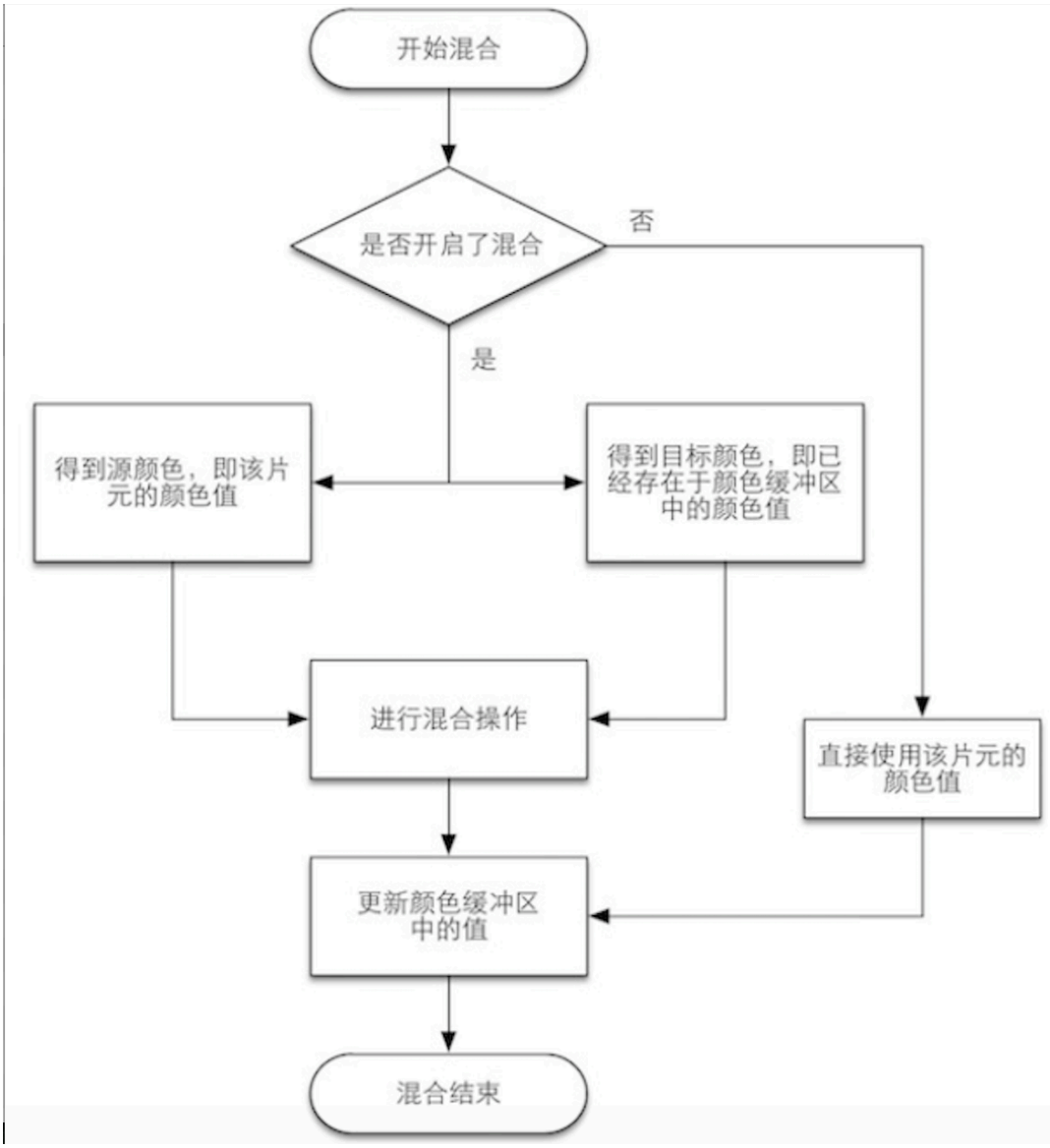
7. 片元着色器

- 片元着色器：又是我们重点关心的。这个阶段是可编程的（废话），上面的阶段给我们准备好了每个片元的所有数据，这个阶段就需要发挥你的能力用这些数据输出片元的颜色了，这里注意的是和前面的顶点着色器处理的单个顶点一样，这里我们处理的是单个片元，不要去想着访问其他片元。这个阶段可以做很多重要的渲染技术，比如纹理采样，用前面插值出来的纹理坐标采样出该片元对应的纹理颜色。

8. 逐片元操作

- 逐片元操作：这个阶段的目的就是进行合并，是高度可配置的。有两个任务，一是经过一系列测试（模板测试、深度测试）决定每个片元的可见性，一是通过测试后留下的片元要和已存在颜色缓存区的颜色进行合并，开启混合的就混合，没开启的就直接覆盖。
- 经过上面的所有计算后就该显示在屏幕上了，GPU 采取多重缓冲的机制，渲染会发生在后置缓冲中，渲染结束后 GPU 可以交换到前置缓冲来显示。


开始模板测试
是否开启了模板测试
否
是
比较参考值（使用读取掩码）和已经存在于模板缓冲区中的模板值（使用读取掩码）
得到模板测试的结果
是否通过了模板测试
否
是
舍弃该片元
根据状态设置更新模板缓冲区中的值（使用写入掩码）
模板测试结束
开始深度测试
是否开启了深度测试
否
是
比较该片元的深度值和已经存在于深度缓冲区中的深度值
得到深度测试的结果
是否通过了深度测试
否
是
是否开启了深度写入
否
是
将深度值写入深度缓冲
舍弃该片元
深度测试结束



2 Unity Shader-渲染队列，ZTest，ZWrite，Early-Z

- https://blog.csdn.net/puppet_master/article/details/53900568

2.1 简介

- 在渲染阶段，引擎所做的工作是把所有场景中的对象按照一定的策略（顺序）进行渲染。最早的是画家算法，顾名思义，就是像画家画画一样，先画后面的物体，如果前面还有物体，那么就用前面的物体把物体覆盖掉，不过这种方式由于排序是针对物体来排序的，而物体之间也可能有重叠，所以效果并不好。所以目前更加常用的方式是 z-buffer 算法，类似颜色缓冲区缓冲颜色，z-buffer 中存储的是当前的深度信息，对于每个像素存储一个深度值，这样，我们屏幕上显示的每个像素点都会进行深度排序，就可以保证绘制的遮挡关系是正确的。而控制 z-buffer 就是通过 ZTest，和 ZWrite 来进行。但是有时候需要更加精准的控制不同类型的对象的渲染顺序，所以就有了渲染队列。今天就来学习一下渲染队列，ZTest，ZWrite 的基本使用以及分析一下 Unity 为了 Early-Z 所做的一些优化。

2.2 Unity 中的几种渲染队列

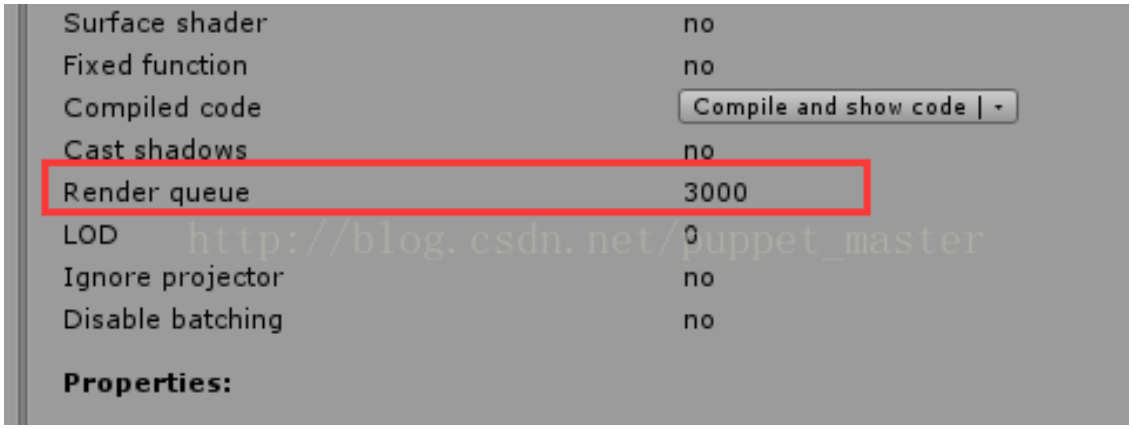
- 首先看一下 Unity 中的几种内置的渲染队列，按照渲染顺序，从先到后进行排序，队列数越小的，越先渲染，队列数越大的，越后渲染。

Background	1000	最早被渲染的物体的队列。
Geometry	2000	不透明物体的渲染队列。大多数物体都应该使用该队列进行渲染，也是 Unity Shader 中默认的渲染队列。
AlphaTest	2450	有透明通道，需要进行 Alpha Test 的物体的队列，比在 Geomerty 中更有效。
Transparent	3000	半透物体的渲染队列。一般是不写深度的物体，Alpha Blend 等的在该队列渲染。
Overlay	4000	最后被渲染的物体的队列，一般是覆盖效果，比如镜头光晕，屏幕贴片之类的。

- Unity 中设置渲染队列也很简单，我们不需要手动创建，也不需要写任何脚本，只需要在 shader 中增加一个 Tag 就可以了，当然，如果不加，那么就是默认的渲染队列 Geometry。比如我们需要我们的物体在 Transparent 这个渲染队列中进行渲染的话，就可以这样写：

```
Tags { "Queue" = "Transparent"}
```

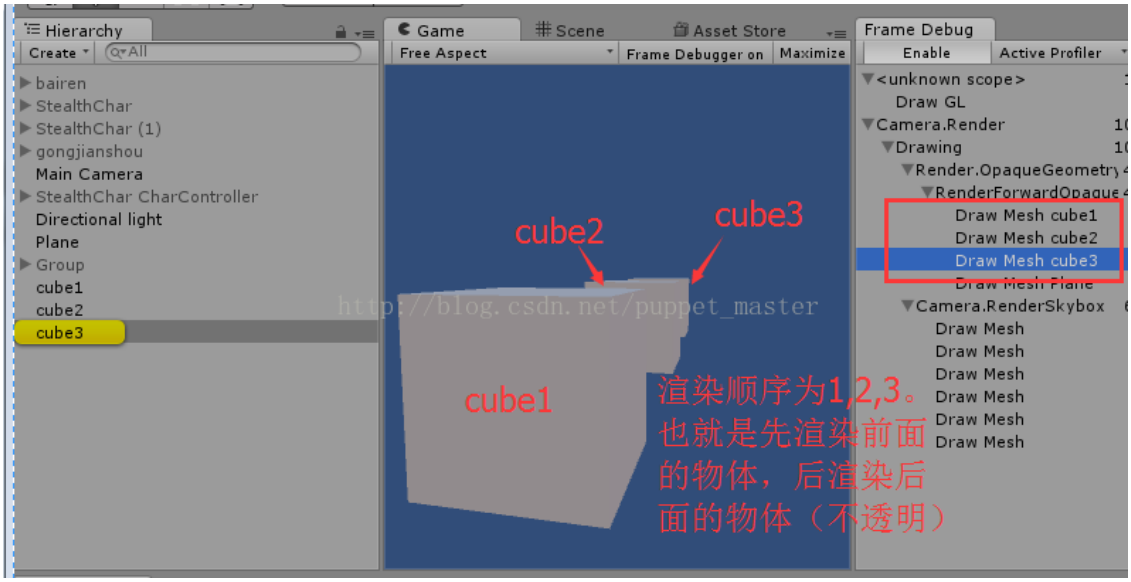
- 我们可以直接在 shader 的 Inspector 面板上看到 shader 的渲染队列：



- 另外，我们在写 shader 的时候还经常有个 Tag 叫 RenderType，不过这个没有 Render Queue 那么常用，这里顺便记录一下：
  - **Opaque**：用于大多数着色器（法线着色器、自发光着色器、反射着色器以及地形的着色器）。
  - **Transparent**：用于半透明着色器（透明着色器、粒子着色器、字体着色器、地形额外通道的着色器）。
  - **TransparentCutout**：蒙皮透明着色器（Transparent Cutout，两个通道的植被着色器）。
  - **Background**：天空盒着色器。
  - **Overlay**：GUITexture，镜头光晕，屏幕闪光等效果使用的着色器。
  - **TreeOpaque**：地形引擎中的树皮。
  - **TreeTransparentCutout**：地形引擎中的树叶。
  - **TreeBillboard**：地形引擎中的广告牌树。
  - **Grass**：地形引擎中的草。
  - **GrassBillboard**：地形引擎何中的广告牌草。

2.3 相同渲染队列中不透明物体的渲染顺序

- 拿出 Unity，创建三个立方体，都使用默认的 bump diffuse shader（渲染队列相同），分别给三个不同的材质（相同材质的小顶点数的物体引擎会动态合批），用 Unity5 带的 Frame Debug 工具查看一下 Draw Call。（Unity5 真是好用得多了，如果用 4 的话，还得用 NSight 之类的抓帧）

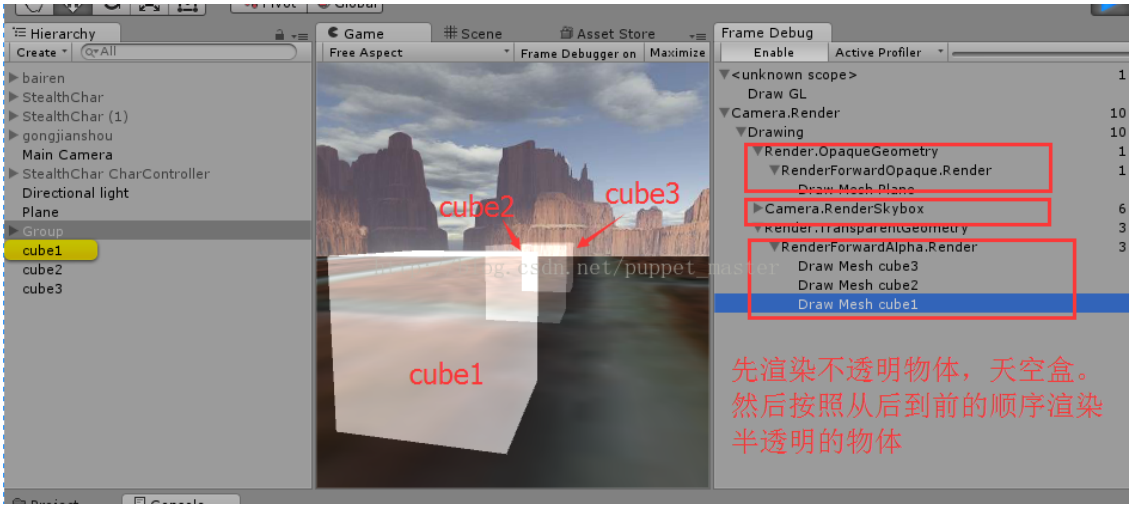


- 可以看出，Unity 中对于不透明的物体，是采用了从前到后的渲染顺序进行渲染的，这样，不透明物体在进行完 vertex 阶段，进行 Z Test，然后就可以得到该物体最终是否在屏幕上可见了，如果前面渲染完的物体已经写好了深度，深度测试失败，那么后面渲染的物体就直接不会再去进行 fragment 阶段。（不过这里需要把三个物体之间的距离稍微拉开一些，本人在测试时发现，如果距离特别近，就会出现渲染次序比较乱的情况，因为我们不知道 Unity 内部具体排序时是按照什么标准来判定的哪个物体离摄像机更近，这里我也就不妄加猜测了）

2.4 相同渲染队列中半透明物体的渲染顺序

- 透明物体的渲染一直是图形学方面比较蛋疼的地方，对于透明物体的渲染，就不能像渲染不透明物体那样多快好省了，因为透明物体不会写深度，也就是说透明物体之间的穿插关系是没有办法判断的，所以半透明的物体在渲染的时候一般都是采用从后向前的方法进行渲染，由于透明物体多了，透明物体不写深度，那么透明物体之间就没有所谓的可以通过深度测试来剔除的优化，每个透明物体都会走像素阶段的渲染，会造成大量的 Over Draw。这也就是粒子特效特别耗费性能的原因。

- 我们实验一下 Unity 中渲染半透明物体的顺序，还是上面的三个立方体，我们把材质的 shader 统一换成粒子最常用的 Particle/Additive 类型的 shader，再用 Frame Debug 工具查看一下渲染的顺序：



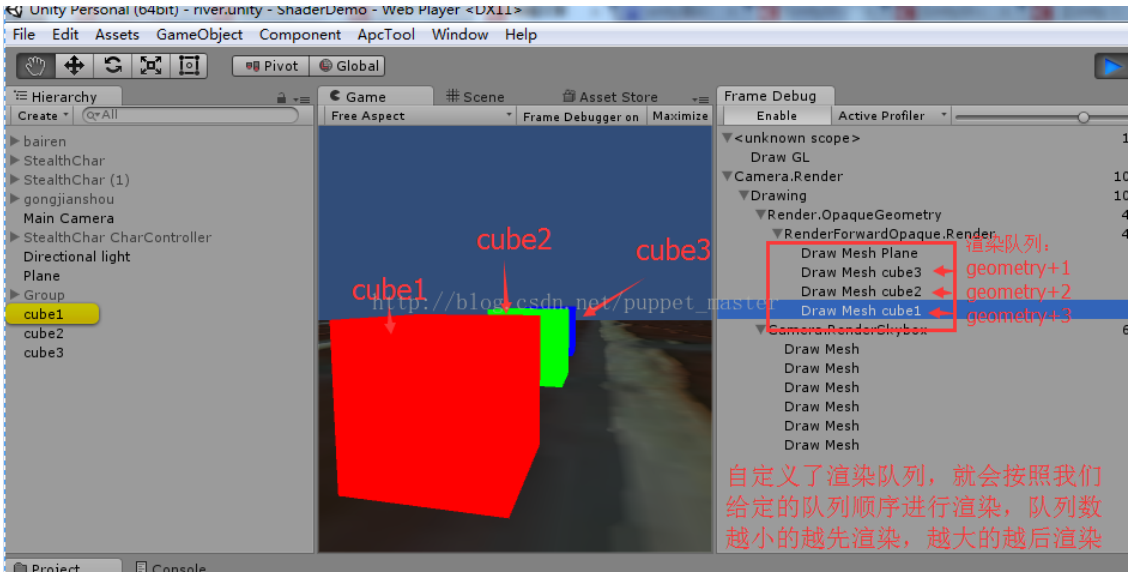
2.5 自定义渲染队列

- Unity 支持我们自定义渲染队列，比如我们需要保证某种类型的对象需要在其他类型的对象渲染之后再渲染，就可以通过自定义渲染队列进行渲染。而且超级方便，我们只需要在写 shader 的时候修改一下渲染队列中的 Tag 即可。比如我们希望我们的物体要在所有默认的不透明物体渲染完之后渲染，那么我们就可以使用 Tag{“Queue”= “Geometry+1”} 就可以让使用了这个 shader 的物体在这个队列中进行渲染。

- 还是上面的三个立方体，这次我们分别给三个不同的 shader，并且渲染队列不同，通过上面的实验我们知道，默认情况下，不透明物体都是在 Geometry 这个队列中进行渲染的，那么不透明的三个物体就会按照 cube1,cube2,cube3 进行渲染。这次我们希望将渲染的顺序反过来，那么我们就可以让 cube1 的渲染队列最大，cube3 的渲染队列最小。贴出其中一个的 shader：

```
Shader "Custom/RenderQueue1" {
 ^^ISubShader {
^^I^^ITags {
            "RenderType"="Opaque" "Queue" = "Geometry+1"}
        Pass {
^^I^^I^^ICGPROGRAM
#pragma vertex vert
#pragma fragment frag
#include "UnityCG.cginc"
            struct v2f {
^^I^^I^^Ifloat4 pos : SV_POSITION;
            };
            v2f vert(appdata_base v) {
^^I^^I^^Iv2f o;
                o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
                return o;
            }
            fixed4 frag(v2f i) : SV_Target {
^^I^^I^^Ireturn fixed4(0,0,1,1);
            }
            ENDCG
        }
    }
    //FallBack "Diffuse"}
}
```

- 其他的两个 shader 类似，只是渲染队列和输出颜色不同。



- 通过渲染队列，我们就可以自由地控制使用该 shader 的物体在什么时机渲染。比如某个不透明物体的像素阶段操作较费，我们就可以控制它的渲染队列，让其渲染更靠后，这样可以通过其他不透明物体写入的深度剔除该物体所占的一些像素。

- PS：这里貌似发现了个问题，我们在修改 shader 的时候一般不需要什么其他操作就可以直接看到修改后的变化，但是本人改完渲染队列后，有时候会出现从 shader 的文件上能看到渲染队列的变化，但是从渲染结果以及 Frame Debug 工具中并没有看到渲染结果的变化，重启 Unity 也没有起到作用，直到我把 shader 重新赋给材质之后，变化才起了效果…（猜测是个 bug，因为看到网上还有和我一样的倒霉蛋被这个坑了，本人的版本是 5.3.2，害我差点怀疑昨天是不是喝了，刚实验完的结果就完全不对了…）

2.6 ZTest（深度测试）和 ZWrite（深度写入）

- 上一个例子中，虽然渲染的顺序反了过来，但是物体之间的遮挡关系仍然是正确的，这就是 z-buffer 的功劳，不论我们的渲染顺序怎样，遮挡关系仍然能够保持正确。而我们对 z-buffer 的调用就是通过 ZTest 和 ZWrite 来实现的。

2.6.1 ZTest

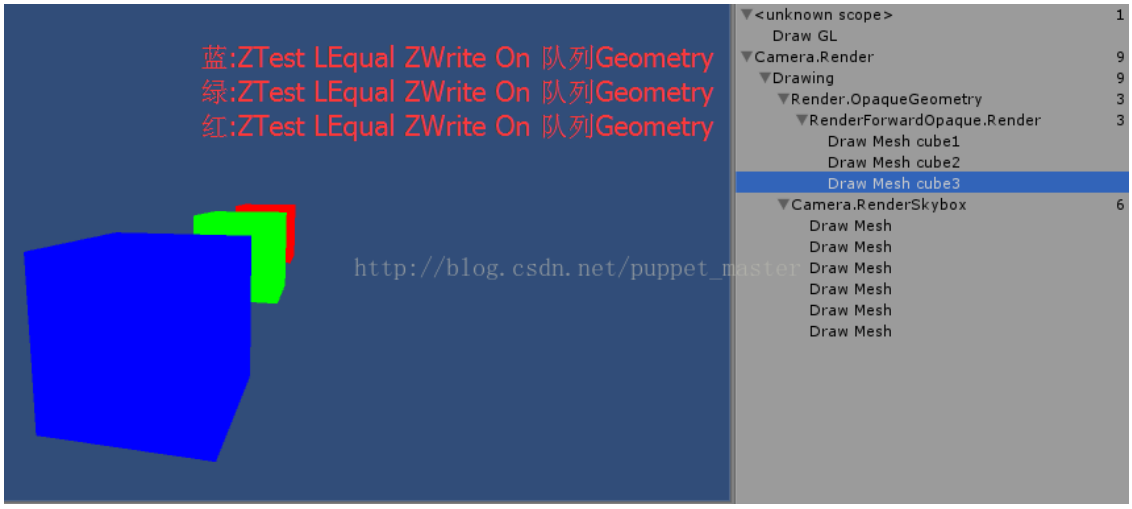
- ZTest 即深度测试，所谓测试，就是针对当前对象在屏幕上（更准确的说是 frame buffer）对应的像素点，将对象自身的深度值与当前该像素点缓存的深度值进行比较，如果通过了，本对象在该像素点才会将颜色写入颜色缓冲区，否则否则不会写入颜色缓冲。ZTest 提供的状态较多。
  - **ZTest Less** （深度小于当前缓存则通过）
  - **ZTest Greater** （深度大于当前缓存则通过）
  - **ZTest LEqual** （深度小于等于当前缓存则通过）
  - **ZTest GEqual** （深度大于等于当前缓存则通过）
  - **ZTest Equal** （深度等于当前缓存则通过）
  - **ZTest NotEqual** （深度不等于当前缓存则通过）
  - **ZTest Always** （不论如何都通过）
  - 注意，*ZTest Off* 等同于 **ZTest Always** ，关闭深度测试等于完全通过。

2.6.2 ZWrite

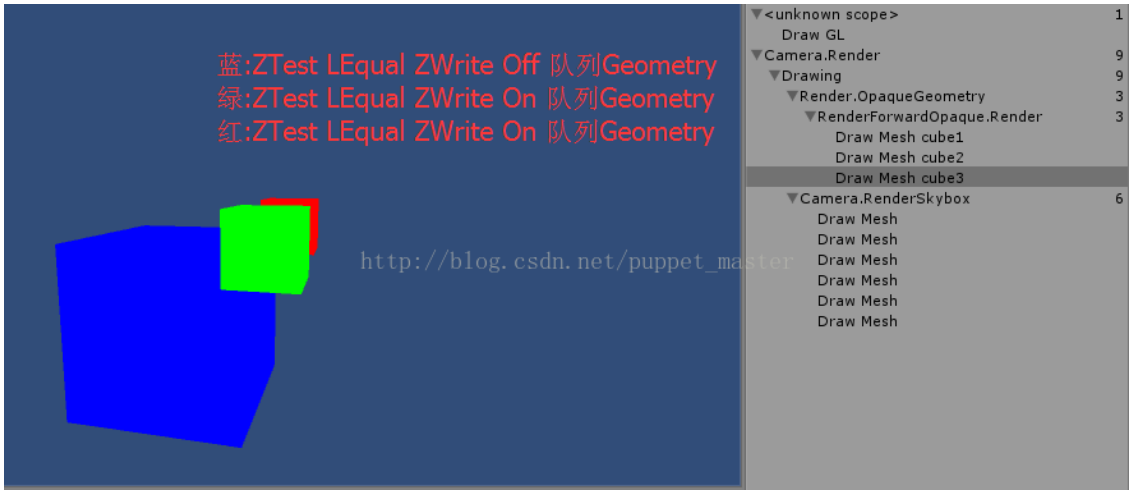
- ZWrite 比较简单，只有两种状态，**ZWrite On** （开启深度写入）和 **ZWrite Off** （关闭深度写入）。当我们开启深度写入的时候，物体被渲染时针对物体在屏幕（更准确地说是 frame buffer）上每个像素的深度都写入到深度缓冲区；反之，如果是 ZWrite Off，那么物体的深度就不会写入深度缓冲区。但是，物体是否会写入深度，除了 ZWrite 这个状态之外，更重要的是需要深度测试通过，也就是 ZTest 通过，如果 ZTest 都没通过，那么也就不会写入深度了。就好比默认的渲染状态是 ZWrite

- On 和 ZTest LEqual，如果当前深度测试失败，说明这个像素对应的位置，已经有一个更靠前的东西占坑了，即使写入了，也没有原来的更靠前，那么也就没有必要再去写入深度了。所以上面的 ZTest 分为通过和不通过两种情况，ZWrite 分为开启和关闭两种情况的话，一共就是四种情况：
  - 1. 深度测试通过，深度写入开启：写入深度缓冲区，写入颜色缓冲区；
  - 2. 深度测试通过，深度写入关闭：不写深度缓冲区，写入颜色缓冲区；
  - 3. 深度测试失败，深度写入开启：不写深度缓冲区，不写颜色缓冲区；
  - 4. 深度测试失败，深度写入关闭：不写深度缓冲区，不写颜色缓冲区；

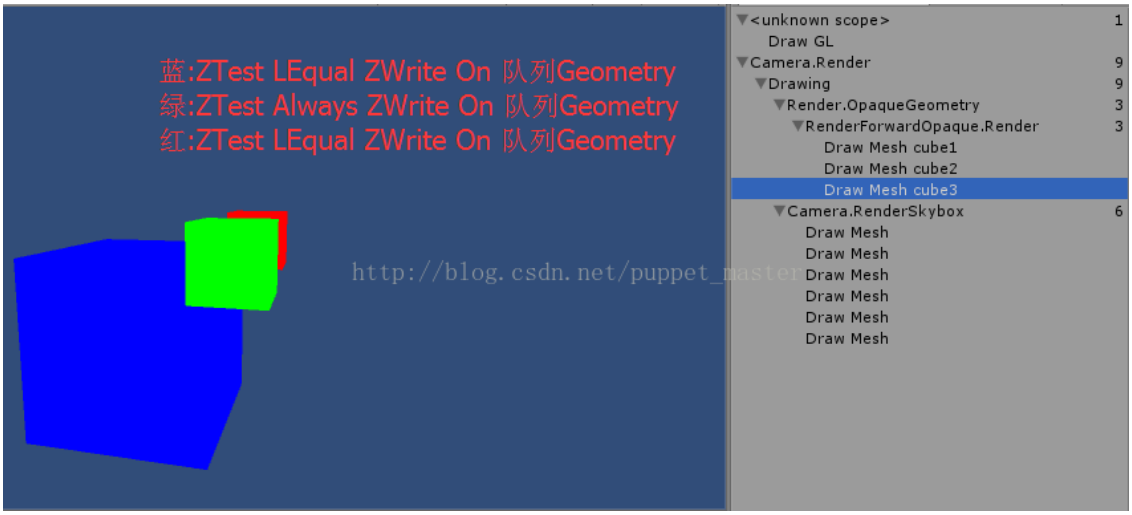
- Unity 中默认的状态（写 shader 时什么都不写的状态）是 ZTest LEqual 和 ZWrite On，也就是说默认是开启深度写入，并且深度小于等于当前缓存中的深度就通过深度测试，深度缓存中原始为无限大，也就是说离摄像机越近的物体会更新深度缓存并且遮挡住后面的物体。如下图所示，前面的正方体会遮挡住后面的物体：



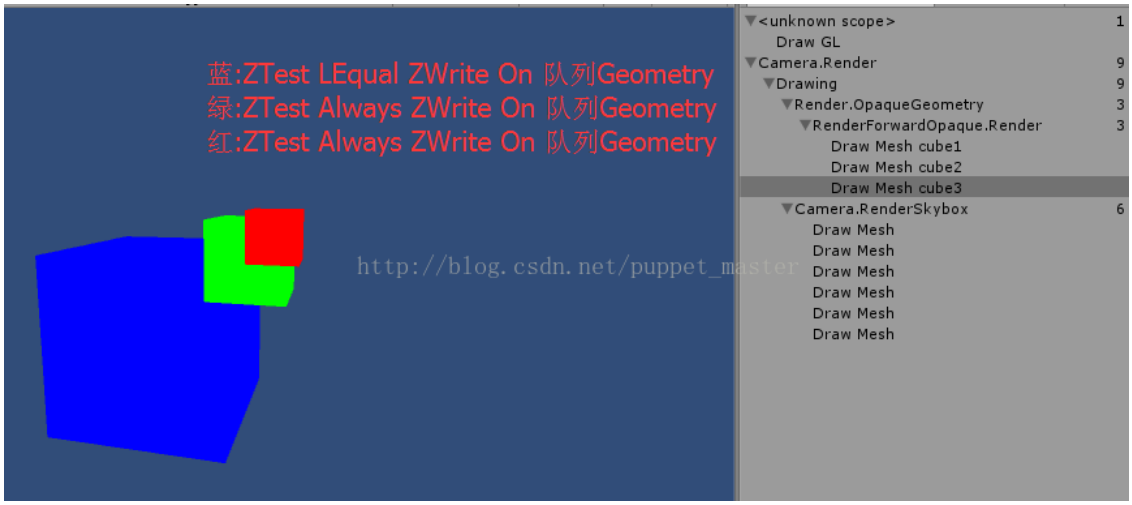
- 写几个简单的小例子来看一下 ZTest，ZWrite 以及 Render Queue 这几个状态对渲染结果的控制。
- 让绿色的对象不被前面的立方体遮挡，一种方式是关闭前面的蓝色立方体深度写入：



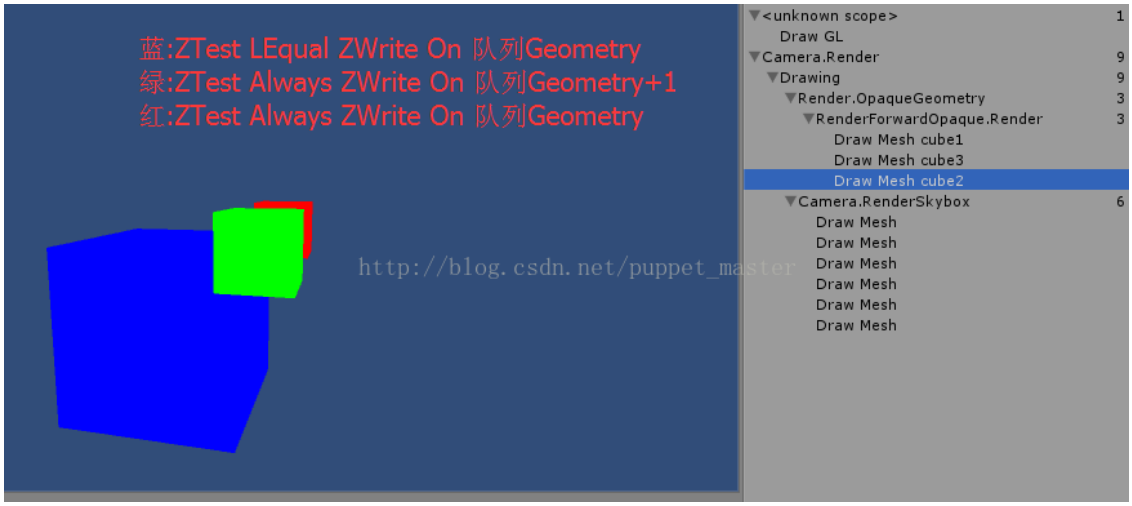
- 通过上面的实验结果，我们知道，按照从前到后的渲染顺序，首先渲染蓝色物体，蓝色物体深度测试通过，颜色写入缓存，但是关闭了深度写入，蓝色部分的深度缓存值仍然是默认的 Max，后面渲染的绿色立方体，进行深度测试仍然会成功，写入颜色缓存，并且写入了深度，因此蓝色立方体没有起到遮挡的作用。
- 另一种方式是让绿色强制通过深度测试：



- 这个例子中其他立方体的 shader 使用默认的渲染方式，绿色的将 ZTest 设置为 Always，也就是说不管怎样，深度测试都通过，将绿色立方体的颜色写入缓存，如果没有其他覆盖了，那么最终的输出就是绿色的了。
- 那么如果红色的也开了 ZTest Always 会怎么样?

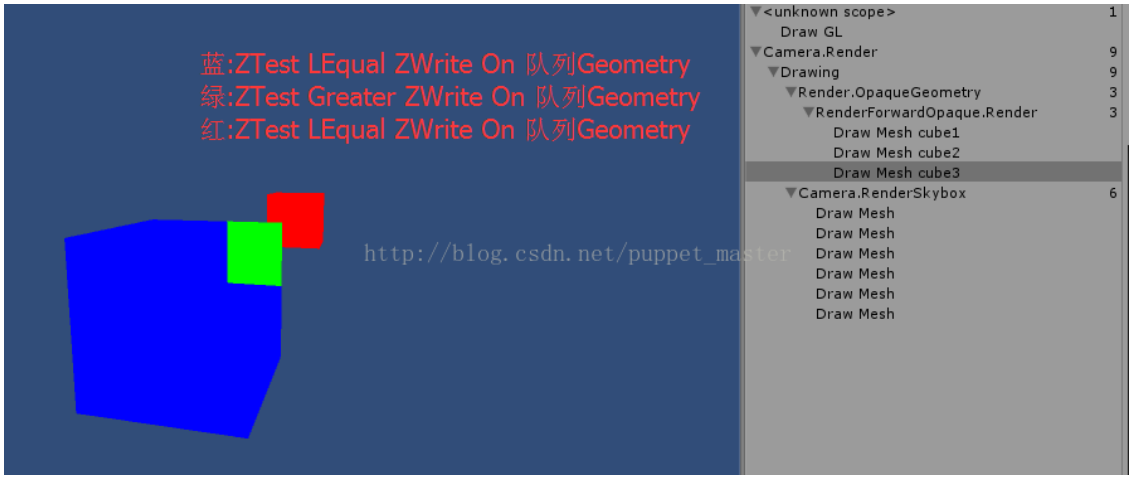


- 在红色立方体也用了 ZTest Always 后，红色遮挡了绿色的部分显示为了红色。如果我们换一下渲染队列，让绿色在红色之前渲染，结果就又不一样了：



- 更换了渲染队列，让绿色的渲染队列 +1，在默认队列 Geometry 之后渲染，最终重叠部分又变回了绿色。可见，当 ZTest 都通过时，上一个写入颜色缓存的会覆盖上一个，也就是说最终输出的是最后一个渲染的对象颜色。

- 再看一下 Greater 相关的部分有什么作用，这次我们其他的都使用默认的渲染状态，绿色的立方体 shader 中 ZTest 设置为 Greater：



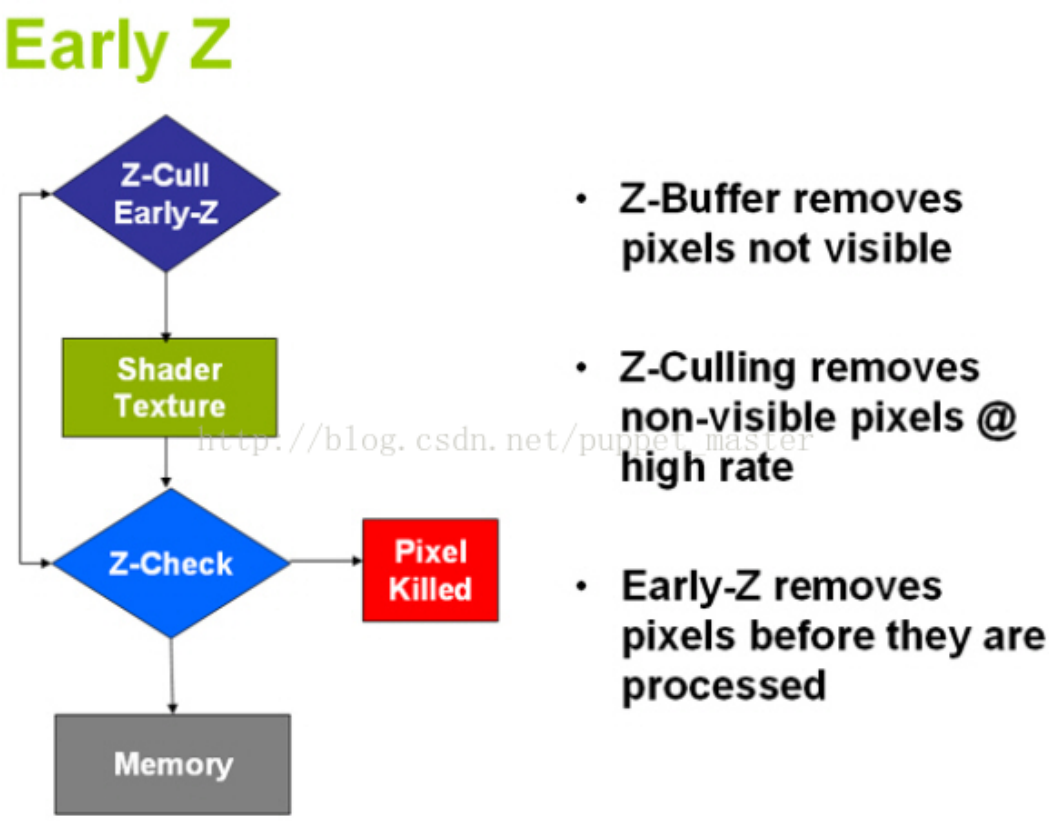
- 这个效果就比较好玩了，虽然我们发现在比较深度时，前面被蓝色立方体遮挡的部分，绿色的最终覆盖了蓝色，是想要的结果，不过其他部分哪里去了呢？简单分析一下，渲染顺序是从前到后，也就是说蓝色最先渲

染，默认深度为 Max，蓝色立方体的深度满足 LEqual 条件，就写入了深度缓存，然后绿色开始渲染，重叠的部分的深度缓存是蓝色立方体写入的，而绿色的深度值满足大于蓝色深度的条件，所以深度测试通过，重叠部分颜色更新为绿色；而与红色立方体重合的部分，红色立方体最后渲染，与前面的部分进行深度测试，小于前面的部分，深度测试失败，重叠部分不会更新为红色，所以重叠部分最终为绿色。而绿色立方体没有与其他部分重合的地方为什么消失了呢？其实是因为绿色立方体渲染时，除了蓝色立方体渲染的地方是有深度信息的，其他部分的深度信息都为 Max，蓝色部分用 Greater 进行判断，肯定会失败，也就不会有颜色更新。

- 有一个好玩的效果其实就可以考 ZTest Greater 来实现，就是游戏里面经常出现的，当玩家被其他场景对象遮挡时，遮挡的部分会呈现出 X-光的效果；其实是在渲染玩家时，增加了一个 Pass，默认的 Pass 正常渲染，而增加的一个 Pass 就使用 Greater 进行深度测试，这样，当玩家被其他部分遮挡时，遮挡的部分才会显示出来，用一个描边的效果渲染，其他部分仍然使用原来的 Pass 即可。

2.7 Early-Z 技术

- 传统的渲染管线中，ZTest 其实是在 Blending 阶段，这时候进行深度测试，所有对象的像素着色器都会计算一遍，没有什么性能提升，仅仅是为了得出正确的遮挡结果，会造成大量的无用计算，因为每个像素点上肯定重叠了很多计算。因此现代 GPU 中运用了 Early-Z 的技术，在 Vertex 阶段和 Fragment 阶段之间（光栅化之后，fragment 之前）进行一次深度测试，如果深度测试失败，就不必进行 fragment 阶段的计算了，因此在性能上会有很大的提升。但是最终的 ZTest 仍然需要进行，以保证最终的遮挡关系结果正确。前面的一次主要是 Z-Cull 为了裁剪以达到优化的目的，后一次主要是 Z-Check，为了检查，如下图：

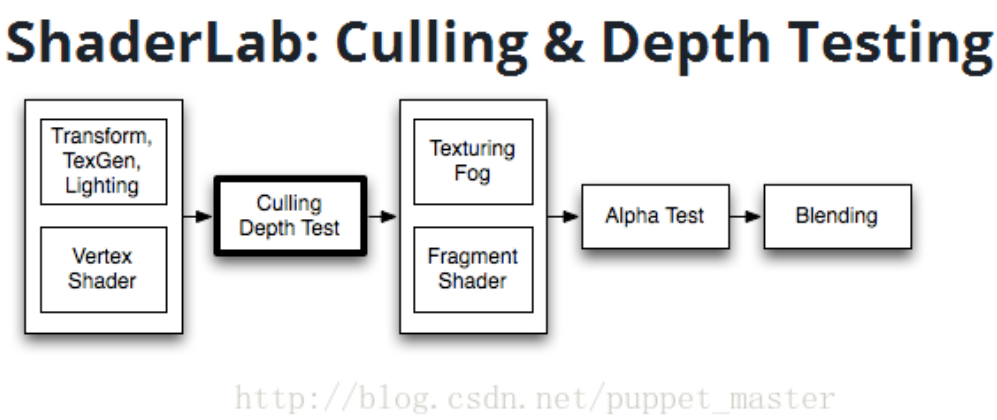


- Early-Z 的实现，主要是通过一个 Z-pre-pass 实现，简单来说，对于所有不透明的物体（透明的没有用，本身不会写深度），首先用一个超级简单的 shader 进行渲染，这个 shader 不写颜色缓冲区，只写深度缓冲区，第二个 pass 关闭深度写入，开启深度测试，用正常的 shader 进行渲染。其实这种技术，我们也可以借鉴，在渲染透明物体时，因为关闭了深度写入，有时候会有其他不透明的部分遮挡住透明的部分，而我们其实不希望他们被遮挡，仅仅希望被遮挡的物体半透，这时我们就可以用两个 pass 来渲染，第一个 pass 使用 Color
- Mask 屏蔽颜色写入，仅写入深度，第二个 pass 正常渲染半透，关闭深度写入。
- 关于 Early-Z 技术可以参考 ATI 的论文 Applications of Explicit Early-Z Culling 以及 PPT，还有一篇 Intel 的文章。

- http://developer.amd.com/wordpress/media/2012/10/ShadingCourse2004_EarlyZ.pdf
- http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/ShadingCourse_ATI.pdf
- https://software.intel.com/en-us/articles/early-z-rejection-sample

2.8 Unity 渲染顺序总结

- 如果我们先绘制后面的物体，再绘制前面的物体，就会造成 over draw；而通过 Early-Z 技术，我们就可以先绘制较近的物体，再绘制较远的物体（仅限不透明物体），这样，通过先渲染前面的物体，让前面的物体先占坑，就可以让后面的物体深度测试失败，进而减少重复的 fragment 计算，达到优化的目的。Unity 中默认应该就是按照最近距离的面进行绘制的，我们可以看一下 Unity 官方的文档中显示的：



Culling is an optimization that does not render polygons facing away from the viewer. All polygons have a front and a back side. Culling makes use of the fact that most objects are closed; if you have a cube, you will never see the sides facing away from you (there is always a side facing you in front of it) so we don't need to draw the sides facing away. Hence the term: Backface culling.

The other feature that makes rendering looks correct is Depth testing. Depth testing makes sure that only the closest surfaces objects are drawn in a scene.

- 从文档给出的流程来看，这个 Depth-Test 发生在 Vertex 阶段和 Fragment 阶段之间，也就是上面所说的 Early-Z 优化。
- 简单总结一下 Unity 中的渲染顺序：**先渲染不透明物体，顺序是从前到后；再渲染透明物体，顺序是从后到前。**

2.9 Alpha Test（Discard）在移动平台消耗较大的原因

- 从本人刚刚开始接触渲染，就开始听说移动平台 Alpha Test 比较费，当时比较纳闷，直接 discard 了为什么会费呢，应该更省才对啊？这个问题困扰了我好久，今天来刨根问底一下。还是跟我们上面讲到的 Early-Z 优化。正常情况下，比如我们渲染一个面片，不管是否是开启深度写入或者深度测试，这个面片的光栅化之后对应的像素的深度值都可以在 Early-Z（Z-Cull）的阶段判断出来了；而如果开启了 Alpha Test（Discard）的时候，discard 这个操作是在 fragment 阶段进行的，也就是说这个面片光栅化之后对应的像素是否可见，是在 fragment 阶段之后才知道的，最终需要靠 Z-Check 进行判断这个像素点最终的颜色。其实想象一下也能够知道，如果我们开了 Alpha Test 并且还用 Early-Z 的话，一块本来应该被剃掉的地方，就仍然写进了深度缓存，这样就会造成其他部分被一个完全没东西的地方遮挡，最终的渲染效果肯定就不对了。所以，如果我们开启了 Alpha Test，就不会进行 Early-Z，Z Test 推迟到 fragment 之后进行，那么这个物体对应的 shader 就会完全执行 vertex shader 和 fragment shader，造成 over draw。有一种方式是使用 Alpha Blend 代替 Alpha Test，虽然也很费，但是至少 Alpha Blend 虽然不写深度，但是深度测试是可以提前进行的，因为不会在 fragment 阶段再决定是否可见，因为都是可见的，只是透明度比较低罢了。不过这样只是权宜之计，Alpha Blend 并不能完全代替 Alpha Test。
- 关于 Alpha Test 对于 Power VR 架构的 GPU 性能的影响，简单引用一下官方的链接以及一篇讨论帖：

When an alpha tested primitive is submitted, early depth testing - such as PowerVR's Hidden Surface Removal (HSR) - can discard fragments that are occluded by fragments closer to the camera. Unlike opaque primitives that would also perform depth writes at this pipeline stage, alpha tested primitives cannot write data to the depth buffer until the fragment shader has executed and fragment visibility is known. These deferred depth writes can impact performance, as subsequent primitives cannot be processed until the depth buffers are updated with the alpha tested primitive's values.

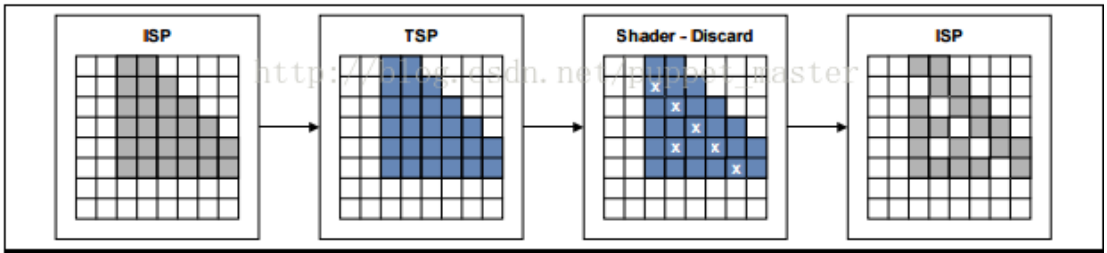


Figure 7. PowerVR alpha test/discard pipeline

2.10 最后再附上两篇参考文章

- http://blog.csdn.net/candycat1992/article/details/41599167
- http://blog.csdn.net/arundev/article/details/7895839

3 基础知识

3.1 Unity 影响渲染顺序因素的总结

- https://blog.csdn.net/u011748727/article/details/68947207

3.1.1 Camera Depth

- 永远最高。Camera Depth 小的一定先进渲染管线。

3.1.2 2、当 Sorting Layer 和 Order In Layer 相同时

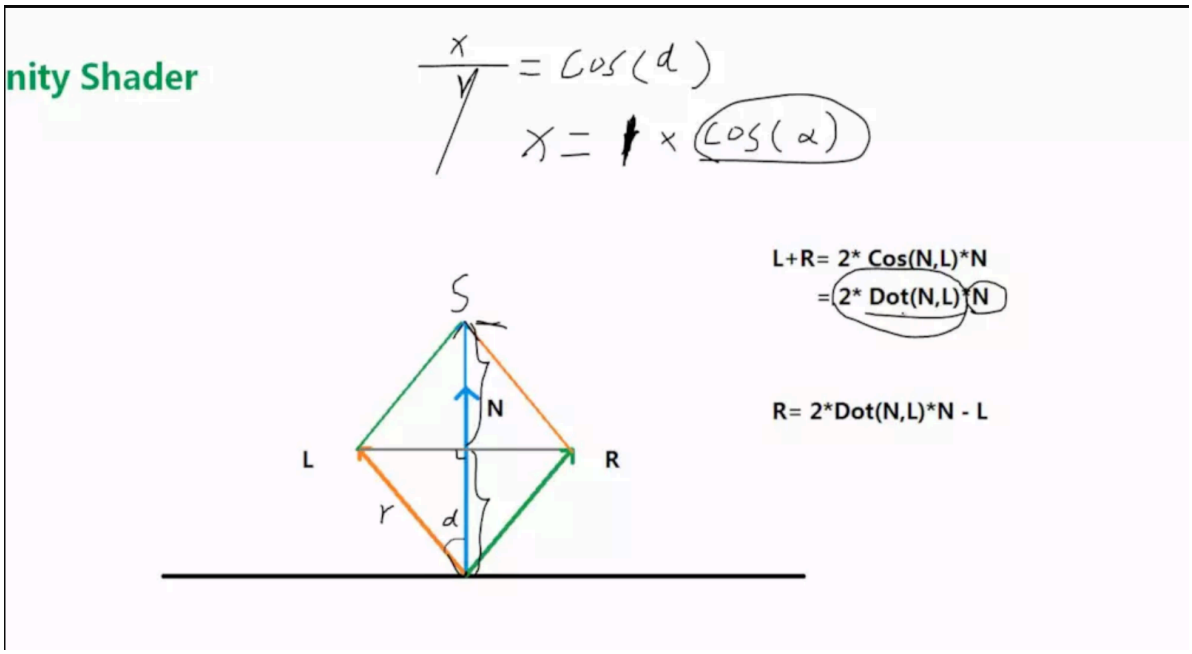
- RenderQueue 小的先进渲染管线。

3.1.3 当 Sorting Layer 和 Order In Layer 不相同时!

- 当两个材质使用了不同的 RenderQueue，且这两个 RenderQueue 都在 [0~2500] 或 [2501~5000] 时，SortingLayer 和 OrderInLayer 的排序生效。
- 当两个材质使用了不同的 RenderQueue，且这两个 RenderQueue 分别在 [0~2500] 和 [2501~5000] 时，则一定会按照 RenderQueue 绘制，无视 SortingLayer、OrderInLayer 的排序。

3.2 specular 镜面反射模型

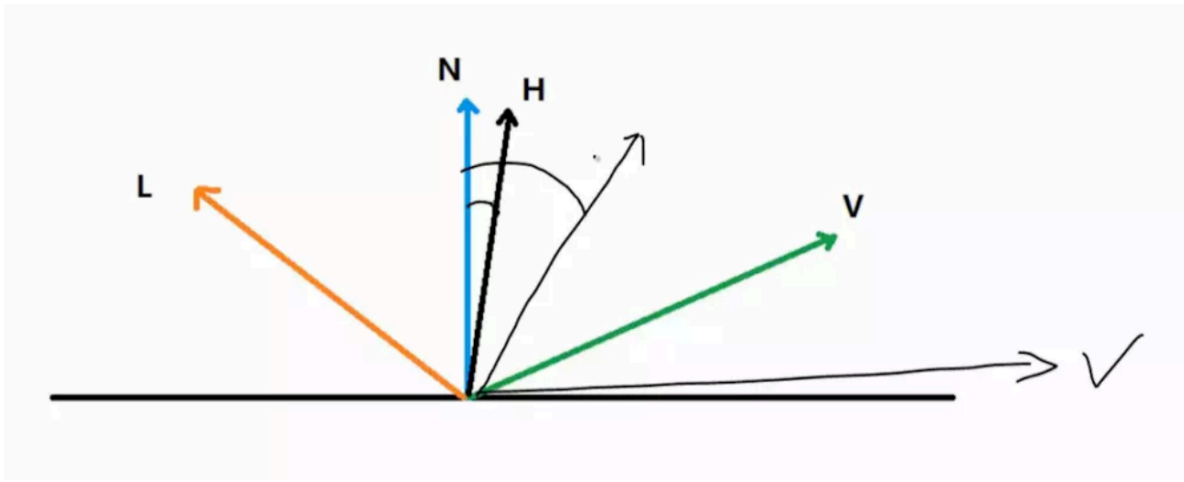
3.2.1 计算反射向量 R 的方法（unityCG 函数库中 reflect 函数可用）



3.2.2 phong 模型，即环境光 + 漫反射 + 镜面反射模型

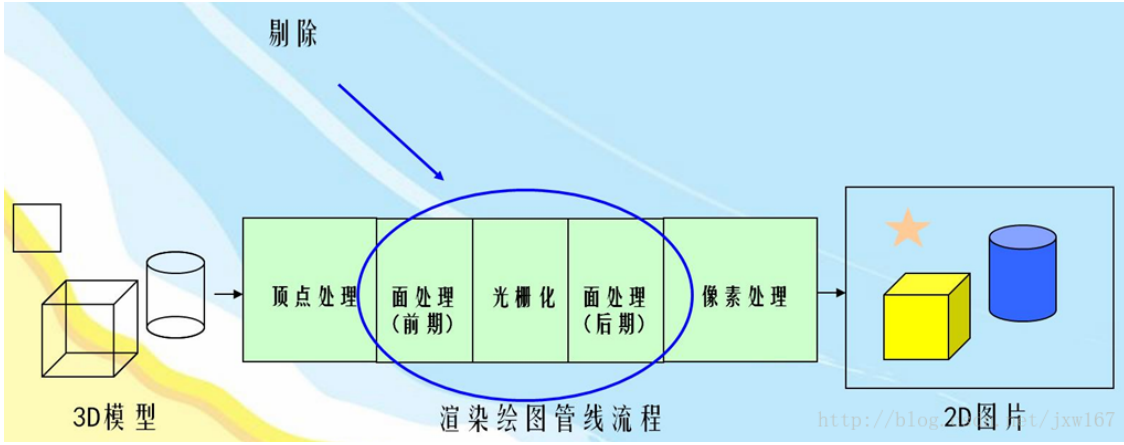
- 漫反射需要一次点积, 镜面反射求反射向量时也要进行一次点积, 但是下面的模型求镜面反射时不需要求反射向量, 而是通过用入射向量 (光) 与平面指向摄像机的向量的和 H 来计算 H 与 N 的夹角, 夹角越小, 镜面反射越强

3.2.3 blinnPhong 模型, 即在 phong 基础上进行一次点积优化的模型

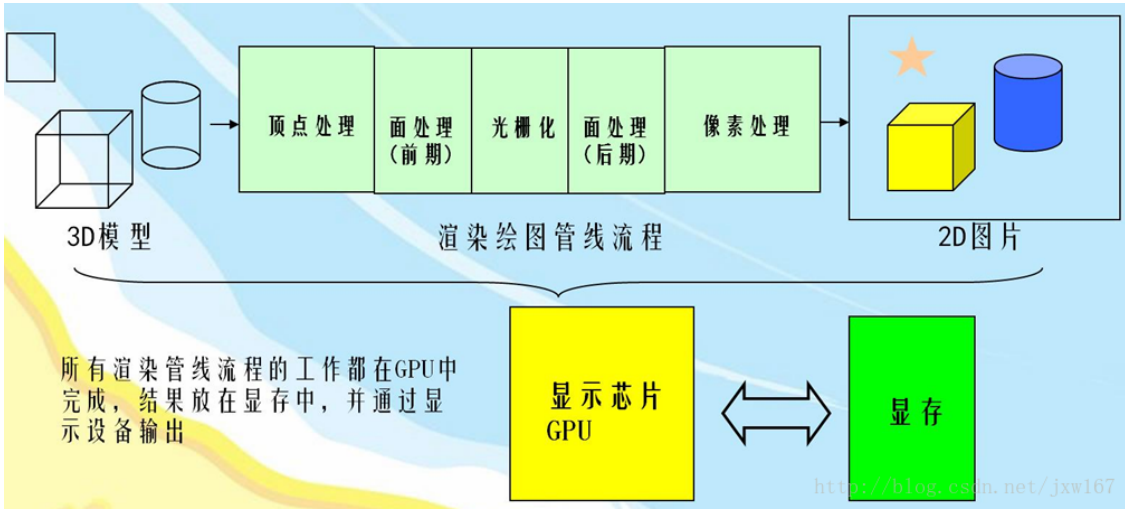


4 详解 Unity3D Shader 开发之渲染管线

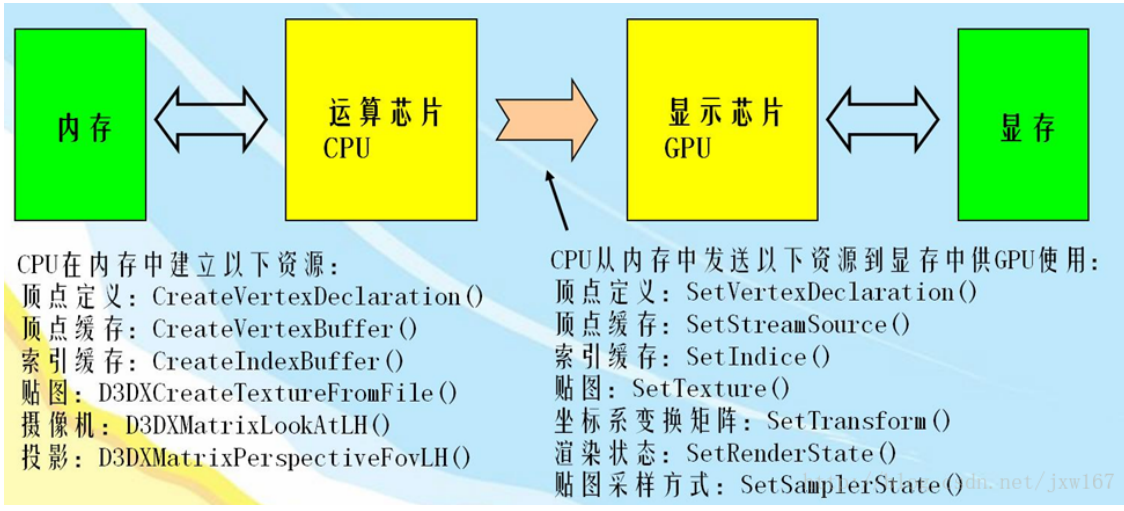
- https://blog.csdn.net/jxw167/article/details/54695181
- 我们需要知道面剔除操作是在渲染管线的哪个部分进行的，将渲染管线中的处理细化一下，效果如下所示：



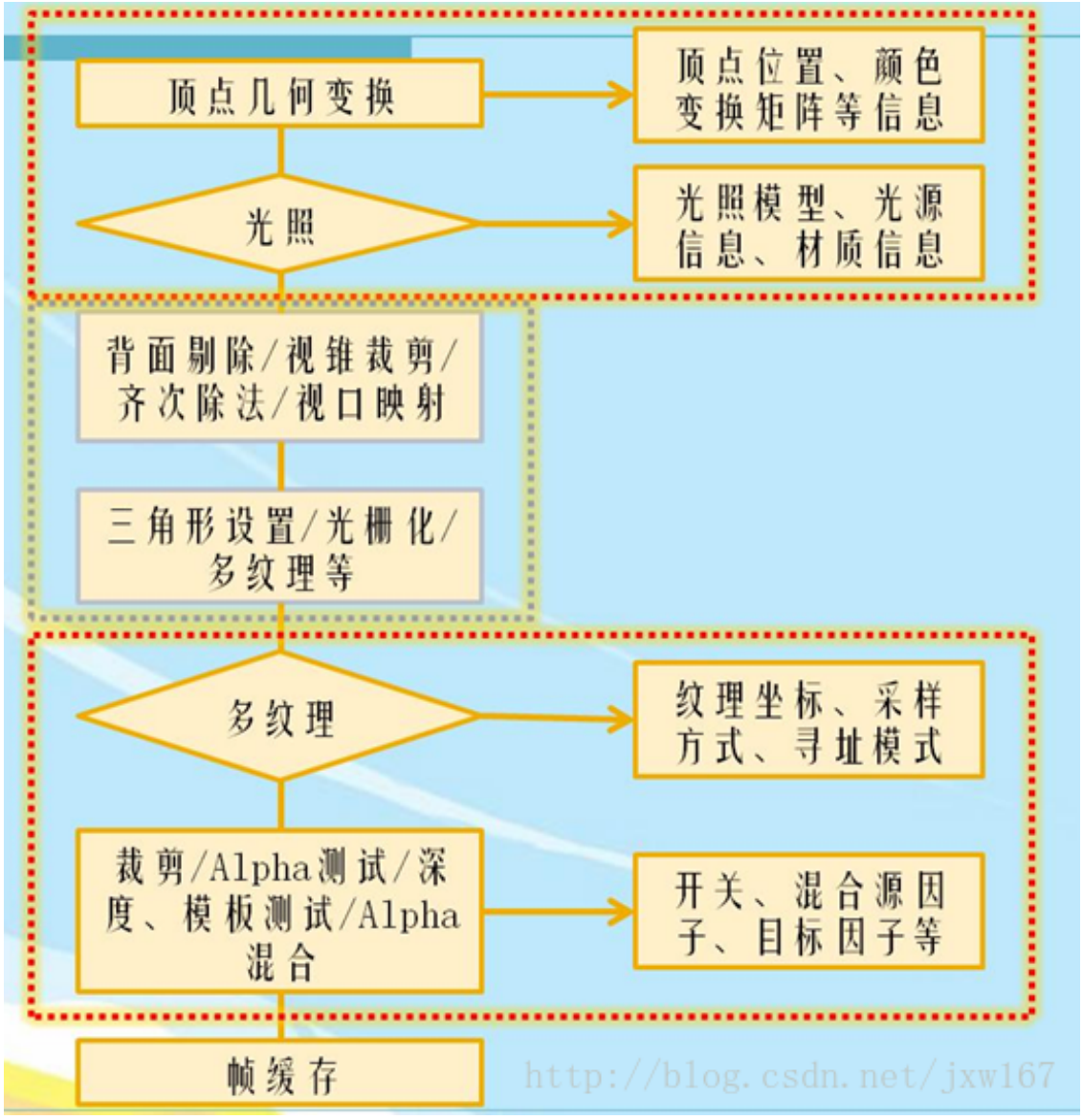
- 渲染管线的流程是在 GPU 中进行的，展示效果如下所示：



- 如果读者使用 DirectX 开发过 Demo，对 3D 调用接口应该比较熟悉，下面结合着图片把在 CPU 中调用的接口对应到 GPU 使用的接口，展示效果如下所示：



- 渲染管线主要分为四个步骤：顶点变换，图元装配，光栅化，像素处理，再结合着图片给读者介绍如下：



- Shader 编程主要是分为两部分：一部分是顶点处理，一部分是像素处理。
  - 顶点处理：顶点渲染的作用是对三维图元的顶点进行坐标变换和光照计算，生成可用于渲染到投影空间的顶点坐标、颜色和纹理坐标。顶点渲染就是定义了一系列针对顶点的渲染指令或渲染语句，当 Direct3D 处理图元顶点时，自动使用这些渲染指令或者渲染语句对每一个顶点逐一进行处理，完成顶点数据的处理工作。
  - 像素处理：对每个像素的颜色斤西瓜混合纹理采样，包括迭代颜色和纹理坐标、纹理采样以及将纹理采样与灯光和材质的颜色进行混合。比如：Alpha 测试、深度测试、模版测试、计算每个像素的雾化值、Alpha 混合等。

5 Unity 移动端性能优化

- https://blog.csdn.net/poem_of_sunshine/article/details/71077171

5.1 渲染

- 利用 reflect probe 代替反射、折射，尽量不用 RTT、GrabPass、RenderWithShader、CommandBuffer.Blit (BuiltinRenderTextureType.CurrentActive...)
- 建立统一后处理框架 (bloom、hdr、DOF 等) 代替多后处理，可以共用模糊函数，减少多次 blit；另外要注意 RTT 的尺寸。

- 空气折射、热浪扭曲等使用 GrabPass 不是所有硬件都支持，改为 RTT 或者后处理来优化。
- 建立统一 shader 材质代替单一 shader，充分利用 shader$_{feature\ multicompile}$
- 图像混合代替多通道纹理，阴影投射、阴影接收、MetaPass、forwardadd 等 pass 不需要时要剔除。
- 少用 alpha test、discard、clip、Alpha Converage 等，因为会影响 Early-Z Culling、HSR 的优化。
- 避免 Alpha Blend 穿透问题（权重混合、深度剥离等透明排序方法代价太大了）。
- 光照贴图代替动态阴影、尽量不用实时光；阴影贴图、环境贴图用 16 位代替 32 位；利用 projector+rtt 或者光圈代替实时阴影。
- 将环境参数（风、雨、太阳）等 shader 全局参数统一管理。
- 非主角可以用 matcap 代替 pbr、无金属不一定要用 pbr，仔细选择物理渲染所用的 FDG（F:schlick、cook-torrance、lerp、要求不高用 4 次方，D：blinn-phong、beckmann、GGX、GGX Anisotropic,G:neumann、cook-torrance、Kelemen、SmithGGX；standard shader 要注意选择 BRDF1-BRDF3），渲染要求不高时不用 GGX；可以用 LH 来优化 GGX。
- 用 fixed、half 代替 float, 建立 shader 统一类型（fixed 效率是 float 的 4 倍，half 是 float 的 2 倍），小心选择 shader 变量的修饰 (uniform、static、全局), 选择 Mobile 或 Unlit 目录下 shader
- 使用高低配渲染，内存足够时可以考虑开启 mipmap
- 使用 surface shader 注意关掉不用的功能，比如：noshadow、noambient、novertexlights、nolightmap、no-dynlightmap、nodirlightmap、nofog、nometa、noforwardadd 等
- standard shader 的变体太多(3 万多)，导致编译时间较长，内存占用也很惊人(接近 1G)，如果使用要关掉没用的 shader$_{feature}$，比如：$PARALLAXMAP\ SHADOWSSOFT\ DIRLIGHTMAPCOMBINED$ DIRLIGHTMAP$_{SEPARA}$
- shaderforge、Amplify Shader Editor 生成的 shader 有多余代码要程序专门优化，Amplify Shader Editor 功能更强大一些，而且开源，建议学习。
- 不要用 unity 自带 terrian，因为即使只用 3 张 splat 图，shader 也是对应 4 个的，建议 T4M 或者转为 mesh。
- 模型和材质相同且数量巨大时用 Instance 来优化，比如草。
- 利用查找纹理 (LUT) 来优化复杂的光照渲染，比如：皮肤、头发、喷漆等。
- 尽量不要使用 Procedural Sky，计算瑞丽散射和米氏散射效率比较低。
- 尽量不要使用 speedtree，改为模型加简单树叶动画，不过 SpeedTreeWind.cginc 里面的动画函数很丰富，TerrianEngine 中的 SmoothTriangleWave 很好用。
- 多用调试工具检查 shader 性能，常用工具有：FrameDebug、Nsight、RenderDoc 、AMD GPU ShaderAnalyzer / PVRShaderEditor、Adreno Profiler 、腾讯 Cube、UWA 等；另外可以内置 GM 界面，比如开关阴影，批量替换 shader 等方便真机调试。

5.2 脚本

- 减少 GetComponent、find 等查找函数在 Update 等循环函数中的调用、go.CompareTag 代替 go.tag 、
- 减少 SendMessage 等同步函数调用；减少字符串连接；for 代替 foreach，5.5 以后版本 foreach 已经优化过了；少用 linq；
- 大资源改为异步加载
- 合理处理协程调用
- 将 AI、网络等放在单独线程
- 发布优化：关闭 log、剔除代码
- 伪随机
- 脚本挂载类改为 Manager 等全局类实现
- lua 中尽量不实现 update、fixedupdate 等循环函数，lua 和 csharp 互调用的效率比较低。

5.3 内存管理

- 池子管理粒子、float UI 等小资源，频繁地 GC 会造成卡顿
- 必要时主动调用 GC.Collect()
- 按照不同资源、不同设备管理资源生命周期，Resources.Load 和 Assetbundle 统一接口，利用引用计数来管理生命周期，并打印和观察生命周期。保证资源随场景而卸载，不常驻内存，确定哪些是预加载，哪些泄漏。
- 内存泄漏（减少驻留内存）：Container 内资源不 remove 掉用 Resources.UnloadUnusedAssets 是卸载不掉的；对于这种情况，建议直接通过 Profiler Memory 中的 Take Sample 来对其进行检测，通过直接查看 WebStream 或 SerializedFile 中的 AssetBundle 名称，即可判断是否存在"泄露"情况；通过 Android PSS/iOS Instrument 反馈的 App 线程内存来查看；
- 堆内存过大：避免一次性堆内存的过大分配，Mono 的堆内存一旦分配，就不会返还给系统，这意味着 Mono 的堆内存是只升不降的。常见：高频调用 new；log 输出；
- CPU 占用高：NGui 的重建网格导致 UIPanel.LateUpdate（按照静止、移动、高频移动来切分）；NGUI 锚点自身的更新逻辑也会消耗不少 CPU 开销。即使是在控件静止不动的情况下，控件的锚点也会每帧更新（见 UIWidget.OnUpdate 函数），而且它的更新是递归式的，使 CPU 占用率更高。因此我们修改了 NGUI 的内部代码，使锚点只在必要时更新。一般只在控件初始化和屏幕大小发生变化时更新即可。不过这个优化的代价是控件的顶点位置发生变化的时候（比如控件在运动，或控件大小改变等），上层逻辑需要自己负责更新锚点。
- 加载用协程；控制同一个 UIPanel 中动态 UI 元素的数量，数量越多，所创建的 Mesh 越大，从而使得重构的开销显著增加。比如，战斗过程中的 HUD 血条可能会大量出现，此时，建议研发团队将运动血条分离成不同的 UIPanel，每组 UIPanel 下 5~10 个动态 UI 为宜。这种做法，其本质是从概率上尽可能降低单帧中 UIPanel 的重建开销。
- 资源冗余：AssetBundle 打包打到多份中；动态修改资源导致的 Instance 拷贝多份（比如动态修改材质，Renderer.meterial，Animation.AddClip）。
- 磁盘空间换内存：对于占用 WebStream 较大的 AssetBundle 文件（如 UI Atlas 相关的 AssetBundle 文件等），建议使用 LoadFromCacheOrDownLoad 或 CreateFromFile 来进行替换，即将解压后的 AssetBundle 数据存储于本地 Cache 中进行使用。这种做法非常适合于内存特别吃紧的项目，即通过本地的磁盘空间来换取内存空间

5.4 美术

- 建立资源审查规范和审查工具：PBR 材质贴图制作规范、场景制作资源控制规范、角色制作规范、特效制作规范；利用 AssetPostprocessor 建立审查工具。
- 压缩纹理、优化精灵填充率、压缩动画、压缩声音、压缩 UI（九宫格优于拉伸）；严格控制模型面数、纹理数、角色骨骼数。
- 粒子：录制动画代替粒子、减少粒子数量、粒子不要碰撞
- 角色：启用 Optimize Game Objects 减少节点, 使用（SimpleLOD、Cruncher）优化面数。
- 模型：导入检查 Read/Write only、Optimize Mesh、法线切线、color、禁用 Mipmap
- 压缩纹理问题：压缩可能导致色阶不足；无透明通道用 ETC1, 现在安卓不支持 ETC2 已不足 5%，建议放弃分离通道办法。
- UI：尽可能将动态 UI 元素和静态 UI 元素分离到不同的 UIPanel 中（UI 的重建以 UIPanel 为单位），从而尽可能将因为变动的 UI 元素引起的重构控制在较小的范围内；尽可能让动态 UI 元素按照同步性进行划分，即运动频率不同的 UI 元素尽可能分离放在不同的 UIPanel 中；尽可能让动态 UI 元素按照同步性进行划分，即运动频率不同的 UI 元素尽可能分离放在不同的 UIPanel 中；
- ugui：可以充分利用 canvas 来切分不同元素。
- 大贴图会导致卡顿，可以切分为多个加载。
- iOS 使用 mp3 压缩、Android 使用 Vorbis 压缩

5.5 批次

- 开启 static batch
- 开启 dynamic batch：要求模型小于 900 顶点，用法线小于 300，用切线小于 180，缩放不一致、使用 lightmap、多通道材质等会使 dynamic batch 无效。
- 减少 GameObject，场景模型数量对 fps 影响巨大。
- 批次不是越少越好，过大的渲染数据会给总线传输带来压力。

5.6 物理

- 不需要移动的物体设为 Static
- 不要用 Mesh 碰撞，角色不用碰撞体
- 触发器逻辑优化
- 寻路频率、AI 逻辑频率、Fixed Timestep、降帧到 30
- 出现卡顿的复杂计算，例如寻路、大量资源加载可以用分帧或者协成异步来处理

6 Unity 性能优化（三）-图形渲染优化

- https://blog.csdn.net/qq_21397217/article/details/80401708

6.1 渲染流程简介

- 在本文中，将会使用“对象”指代游戏中需要被渲染的对象，任何含有 Renderer 组件的 GameObject 都会被称为对象。
- 通常使用渲染管线来描述渲染流程，其过程可以大致描述为：
  - CPU 决定哪些事物需要绘制以及它们如何被绘制。
  - CPU 向 GPU 发送指令。
  - GPU 根据 CPU 的指令对事物进行绘制。
- 每渲染一帧画面，CPU 都会进行如下工作：
  - CPU 检查场景中的每个对象来确定它是否需要进行渲染。只有满足指定条件的对象才会被渲染，例如：对象必须在相机的视锥体（Frustum）内并且没有被剔除（Culling）时才会被渲染。
  - CPU 将每个需要渲染的对象的 Mesh 渲染数据编排到 Draw Call 指令中。某些情况下，一些共享配置的对象可能被合并到同一个 Draw Call 中，这个过程称为批处理（Batching），参考 Draw Call 批处理手册。
  - CPU 为每次 Draw Call 创建一个称为 Batch 的数据包。
- 每次 Draw Call，CPU 要执行下列操作：
  - CPU 可能会向 GPU 发送 SetPass Call 指令来修改一些被统称为 Render State 的变量。每个 SetPass Call 都会告知 GPU 在下次渲染 Mesh 时要使用哪个配置。只有在下次需要渲染的 Mesh 的 Render State 与当前的 Render State 不同时，才会有 SetPass Call。
  - CPU 向 GPU 发送 Draw Call 指令。Draw Call 指令告知 GPU 使用最近一次的 SetPass Call 的配置对指定的 Mesh 进行渲染。
  - 在某些情况下，一个 Batch 可能需要不止一个 Pass。Pass 是一段 Shader 代码，新的 Pass 会改变 Render State。CPU 必须为 Batch 中的每个 Pass 发送新的 SetPass Call 并再次发送 Draw Call。
- 与此同时，GPU 进行着如下的工作：
  - GPU 根据 CPU 的指令执行任务，执行顺序与指令发送顺序相同。
  - 如果当前的任务是一个 SetPass Call，则 GPU 更新 Render State。
  - 如果当前的任务是一个 Draw Call，则 GPU 渲染 Mesh。
  - 重复上述过程直到 GPU 将来自 CPU 的指令全部处理完。

6.2 渲染问题的类型

- 引起渲染问题的基本原因有两个：
  - 渲染管线（Rendering Pipeline）效率低，渲染管线中的一步或多步操作花费过多时间将导致数据流通不畅，这些低效问题被称为瓶颈（Bottleneck）。
  - 在渲染管线中加入了太多数据，即使是在效率最高的渲染管线中，也有对一帧中处理的数据量的限制。
- 如果游戏中因为 CPU 执行渲染任务耗时太久而导致帧渲染时间过长，我们称其为 CPU 受限（CPU Bound）。
- 如果游戏中因为 GPU 执行渲染任务耗时太久而导致帧渲染时间过长，我们称其为 GPU 受限（GPU Bound）。

6.3 处理 CPU Bound

- 通常将帧渲染过程中的必须由 CPU 处理的任务分为三类：
  - 决定什么必须被绘制
  - 为 GPU 准备指令
  - 向 GPU 发送指令
- 这几个大类中包含很多独立的任务，这些任务可能被分散到多个线程中同时执行。在多个线程中执行渲染任务被称为多线程渲染（Multithreaded Rendering）。
- 在 Unity 的渲染过程中涉及了三中类型的线程：主线程（Main Thread）、渲染线程（Render Thread）和工作线程（Worker Thread）。主线程执行游戏中的大多数 CPU 任务，包括一些渲染任务；渲染线程专门用于向 GPU 发送指令；工作线程每次执行一个任务，例如剔除（Culling）或者蒙皮（Mesh Skining），CPU 核数越多，可以创建的工作线程就越多。
- 并不是所有的平台都支持多线程渲染，比如，目前的（Unity 5.4）WebGL 平台就不支持多线程渲染。在不支持多线程渲染的平台上，所有的 CPU 任务都在同一个线程中处理。

6.3.1 使用图形作业（Graphics Jobs）功能

- Player Settings 中的 Graphics jobs 选项可以让 Unity 将那些本该由主线程处理的渲染任务分配到工作线程中，有些时候也会把渲染线程的任务分配到工作线程。在可以使用该功能的平台上，它将带来显著的性能提升。该功能目前（Unity 2018.1.0）仍处于试验阶段，可能造成游戏崩溃。

6.3.2 减少向 GPU 发送的数据量

- 向 GPU 发送指令的时间开销是引起 CPU Bound 的最常见原因。该过程在大多数平台上都会分配到渲染线程中执行，但在某些平台（比如 PlayStation 4）上会分配到工作线程中执行。
- 向 GPU 发送指令的过程中，开销最大的操作是 SetPass Call。如果游戏在向 GPU 发送指令的过程中发生了 CPU Bound，那么降低 SetPass Call 数量可能是最好的提升性能方式。通常，降低 Batch 数目、让更多的对象共享相同的 Render State 会降低 SetPass Call 数目，进而提高 CPU 性能。即使降低 Batch 数目没能够让 SetPass Call 数目降低，它也能提升性能，因为 CPU 处理单个 Batch 的能力比处理多个 Batch 的能力更强，即使这些 Batch 中包含的 Mesh 数据量是相同的。
- 通常，有 3 种降低 Batch 和 SetPass Call 数目的方法：
  - 降低需要渲染的对象的数量，可以同时减少 Batch 和 SetPass Call。
  - 降低每个对象需要被渲染的次数，可以减少 SetPass Call。
  - 将对象合并到更少的批处理当中，可以减少 Batch。

6.3.3 降低需要渲染的对象的数量

- 减少 Scene 中可见的对象数量。
- 缩短相机绘制距离。使用相机的 Far Clip Plane 属性控制相机的绘制距离，可以使用雾（Fog）掩饰远处缺失的对象。
- 在代码中设置相机的 Layer Cull Distances 属性来更细致地控制相机在不同层（Layer）地绘制距离。
- 使用相机地遮挡剔除（Occlusion Culling）选项关闭对被其他对象遮挡地对象地渲染。遮挡剔除并不适用于所有 Scene，它会造成额外的 CPU 开销。这篇博客讲述了更多关于遮挡剔除地内容。

6.3.4 降低每个对象需要被渲染的次数

- 实时光照、阴影和反射可以增强游戏地真实性，但使用这些特性会导致对象被渲染多次，很影响性能。这些特性所造成地具体影响取决于游戏所使用的渲染通道（Rendering Path）。渲染通道是指绘制 Scene 时的运算的执行顺序，不同渲染通道间的主要区别是处理实时光照、阴影和反射的方式。通常来说，运行在高性能机器上并且使用了较多实时光照、阴影和反射的游戏适合使用 Deferred Rendering，而运行在低性能机器上并且没有使用这些特性的游戏适合使用 Forward Rendering。
- 使用烘培（Baking）预先计算那些不会发生变化的对象的光照信息，减少实时光照的计算量。
- 在 Quality Settings 中调整阴影属性，控制阴影质量。
- 尽量少用反射探针（Reflection Probe），它们会导致 Batch 数量大量增加。
- Unity 手册中对渲染通道进行了更多讲解。
- 光照与渲染介绍中对 Unity 的光照进行了详细的讲解。

6.3.5 将对象数据合并到更少的 Batch 当中

- 关于批处理优化的详细介绍，参考开头提到的那篇文章：【Unity 技巧】Unity 中的优化技术。
- 一个批处理中可以包含多个对象的数据，前提是这些对象满足下列条件：
  - 共用同一个材质（Material）的同一个实例。
  - 拥有相同的材质设置（例如：纹理、Shader 和 Shader 参数）。
- 批处理可以提升性能，但是要小心使用，以免批处理过程的开销比其节省的开销还要高。
- 对于满足批处理条件的对象，有几种不同的优化技术：
  - **静态批处理（Static Batching）**：用于处理临近的静态对象（被标记为 Batching Static），会占用更多的内存。
  - **动态批处理（Dynamic Batching）**：用于处理非静态对象，它的限制比较多，而且会占用更多的 CPU 时间。
  - **UI 批处理**：对 UI 的批处理比较复杂，它受 UI 布局的影响，请参考 Unity UI 优化指导。
  - **GPU 实例（GPU Instancing）**：用于处理集中出现的大量的独立对象，例如粒子。GPU 实例的限制比较多，而且需要硬件支持，请参考 GPU 实例手册。
  - **纹理图集（Texture Atlasing）**：将多个纹理贴图合并到一张大的纹理贴图中。
  - **手动合并 Mesh**：通过 Unity 编辑器或者代码将共用相同材质和纹理的 Mesh 合并。手动合并 Mesh 会影响对象的剔除，请参考 Mesh.CombineMeshes。
- 在代码中要谨慎使用 Renderer。material，他会将 material 对象复制一份并非返回新对象的引用。这会破坏对象的批处理条件，因为 Renderer 不再持有与其他对象共用的材质的实例。如果要在代码中获取批处理对象的材质，应该使用 Renderer.sharedMaterial，参考 Draw Call 批处理手册。

6.3.6 剔除、排序和批处理优化

- 剔除、获取需要渲染的对象的数据、将数据排序到批处理中以及生成 GPU 指令都能导致 CPU 受限。这些任务可以在主线程或者独立的工作线程中执行，具体取决于游戏设置和目标硬件平台。
  - 剔除操作本身的开销不算太高，但减少不必要的剔除仍可以提升性能。Scene 中的每个激活（active）的相机和对象都会产生开销，应该禁用（disable）那些当前不适用的相机和 Renderer。
  - 批处理可以极大的提升向 GPU 发送指令的速度，但有时他可能造成意想不到的开销。如果批处理造成了 CPU 受限，就应该限制游戏中手动/自动批处理的数量。

6.3.7 蒙皮（Skinned Meshes）优化

- SkinnedMeshRenderer 组件用于处理骨骼动画，常用在角色动画上。与蒙皮相关的任务可以在主线程或者独立的工作线程中执行，具体取决于游戏设置和目标硬件平台。
- 蒙皮渲染的开销比较高，下面一些对蒙皮渲染进行优化的手段：
  - 减少 SkinnedMeshRenderer 组件的数量。导入模型时，模型可能带有 SkinnedMeshRenderer 组件，如果游戏中该模型并不会使用骨骼动画，就应该将 SkinnedMeshRenderer 组件替换为 MeshRenderer。在导入模型时，可以选择不导入动画，请参考模型导入设置。
  - 减少使用 SkinnedMeshRenderer 的对象的 Mesh 顶点数，参考蒙皮渲染器手册。
  - 使用 GPU Skinning。在硬件平台支持并且 GPU 资源足够的条件下，可以在 Player Setting 中启用 GPU Skinning，将蒙皮任务从 CPU 转移到 GPU。
- 更多优化内容请参考角色模型优化手册。

6.3.8 减少主线程中与渲染无关的操作

- 很多与渲染无关的任务都主线程中执行，减少主线程中与渲染无关的任务的 CPU 时间消耗。

6.4 处理 GPU Bound

6.4.1 优化填充率（Fill Rate）

- 填充率指 GPU 每秒能够渲染的屏幕像素数量，它是引起 GPU 的性能问题的最常见因素，尤其是在移动设备上。下面列出一些与填充率相关的优化手段：
  - 优化 片元着色器（**Fragment Shader**） 。片元着色器是用于控制 GPU 如何绘制单个像素的 Shader 代码段，绘制每个像素点都需要执行这段代码。复杂的片元着色器是引起填充率问题的常见原因。
    * 如果在使用 Unity 内置 Shader，优先使用能够满足视觉效果要求的最简单高效的 Shader。例如，Unity 内置的几种适用于移动平台的 Mobile Shader 都经过高度优化，而且它们也可以于移动平台以外的其他平台，如果它们提供的视觉效果能够满足需求，就应该使用这些 Shader。
    * 如果在使用 Unity 提供的 Standard Shadar，Unity 会根据当前的材质设置对 Shader 进行编译，只有当前使用了的属性才会被编译。这意味着，移除材质中的一些可以省略的属性（例如 Secondary Maps(Detail Maps)）能够提升性能。

如果在使用自行编写的 Shader，应该尽可能对其进行优化，请参考 Shader 优化提示。

- 减少 重绘（**Overdraw**）。重绘指对同一个像素点进行多次绘制，这种情况发生在对象被绘制在其他对象上层时，它会极大地增加填充率。最常见地引起过渡重绘地因素是透明材质、未经优化地粒子和重叠地 UI 元素，应该对这些内容进行优化或者减少使用量。请参考：渲染队列、子着色器标签和填充率、画布和输入。
- 减少 图像特效（**Image Efficts**） 。图像特效是引起填充率问题的重要因素，尤其是在使用了多个图像特效时。在同一个相机上使用多个图像特效会产生多个 Shader Pass，将多个图像特效的 Shader 代码合并到单个 Pass 中可以提升性能，参考后期处理。

6.4.2 优化显存带宽（Memory Bandwidth）

- 显存带宽指 GPU 对其专用的内存的读写速率。如果游戏出现带宽受限，通常是因为使用了太大的纹理（Texture）。
- 优化纹理的方法：
  - 纹理压缩（Texture Compression）可以极大的减少内存和磁盘占用。
  - Mipmap 是用在远处的物体上的低分辨率版本纹理。如果游戏中需要显示离相机很远的物体，则可以使用 Mipmap 减少显存带宽占用。
- 纹理优化的具体细节，请参考纹理手册。

6.4.3 优化顶点处理（Vertex Processing）

- 顶点处理指 GPU 渲染 Mesh 中的每个顶点使所做的工作。顶点处理的开销与两个因素有关：需要进行渲染的顶点的数量和对每个顶点所要进行的操作。
- 优化顶点处理方法：
  - 降低 Mesh 复杂度。
  - 使用 **法线贴图（Normal Map）** ，请参考法线贴图手册。
  - 禁用不使用法线贴图的模型的顶点切线，这可以减少每个顶点发送给 GPU 的数据量，请参考模型导入设置。
  - 使用 **LOD** ，让物体根据离相机的距离展示不同的细节，请参考 LOD 组手册。
  - 降低 **顶点着色器（Vertex Shadar）** 代码复杂度。顶点着色器是用于控制 GPU 如何绘制每个顶点的 Shader 代码段，优化方式请参考片元着色器优化方式的第 1、3 条。

7 Unity 技巧】Unity 中的优化技术

- https://blog.csdn.net/candycat1992/article/details/42127811