Unity 的渲染路径

在 Unity 里，**渲染路径（Rendering Path）**决定了光照是如何应用到 Unity Shader 中的。因此，如果想要和光源打交道，我们就需要为每个 Pass 指定它使用的渲染路径。也就是说，我们只有为 Shader 正确地选择和设置了需要的渲染路径，该 Shader 的光照计算才能被正确执行。

我们可以在每个 Pass 中使用标签来指定该 Pass 使用的渲染路径。这是通过设置 Pass 的 `LightModel` 标签实现的。不同类型的渲染路径可能会包含多种标签设置。下面给出了 Pass 的 `LightModel` 标签支持的渲染路径设置选项。

`Always` 不管使用哪种渲染路径，该 Pass 总是会被渲染，但不会计算任何光照。

`ForwardBase` 用于**前向渲染**，该 Pass 会计算环境光，最重要的平行光，逐顶点 / SH 光源和 Lightmaps。

`ForwardAdd` 用于**前向渲染**，该 Pass 会计算额外的逐像素光源，每个 Pass 对应一个光源。

`Deferred` 用于**延迟渲染**，该 Pass 会渲染 G 缓冲（G-buffer）。

`ShadowCaster` 把物体的深度信息渲染到阴影映射纹理（shadowmap）或一张深度纹理中。

那么指定渲染路径到底有什么用呢？如果一个 Pass 没有指定任何任何渲染路径会有什么问题吗？我们来看看 Unity 的渲染引擎是如何处理这些渲染路径的吧。

前向渲染路径

前向渲染路径是传统的渲染方式，也是我们最常用的一种渲染路径。

前向渲染路径的原理

每进行一次完整的前向渲染，我们需要渲染该对象的渲染图元，并计算颜色缓冲区和深度缓冲区的信息。我们利用深度缓冲区来决定一个片元是否可见，如果可见就更新颜色缓冲区中的颜色值。我们可以用伪代码来描述前向渲染路径的大致过程：

```
Pass {
    for ( each primitive in this model ) {
        for ( each fragment covered by this primitive ) {
            if ( failed in depth test ) {
                // 如果没有通过深度测试,说明该片元不可见.
                discard;
            } else {
                // 如果该片元可见,就计算光照
                float4 color = Shadering(materialInfo, pos, normal, lightDir, viewDir);
                // 更新帧缓冲区
                writeFrameBuffer( fragment, color );
            }
        }
    }
}
```

对于每个逐像素光源，我们都需要进行一次上面完整的渲染流程 . 如果一个物体在多个逐像素光源的影响区域内，那么该物体就需要执行多个 Pass，每个 Pass 计算一个逐像素光源的光照结果，然后在帧缓冲中把这些光照结果混合起来得到最终的颜色值 . 看一个例子 : 假设场景中有 N 个物体，每个物体受到 M 个光源的影响，那么要渲染整个场景一共需要 M * N 个 Pass . 可以看出，如果有大量逐像素光照，那么需要执行的 Pass 数目也会很大 . 因此，渲染引擎通常会限制每个物体的逐像素光照数目 .

Unity 中的前向渲染

实际上，一个 Pass 不仅可以用来计算逐像素光照，也可以用来计算逐顶点等其他光照 . 这取决于光照计算所处流水线阶段以及计算时使用的数学模型 . 当我们渲染一个物体时，Unity 会计算哪些光源照亮了它，以及这些光源照亮该物体的方式 .

在 Unity 中，前向渲染路径有 3 种处理光照 (即照亮物体) 的方式 : **逐顶点处理，逐像素处理，球谐函数 (Spherical Harmonics，SH) 处理** . 而决定一个光源使用哪种处理模式取决于它的**类型**和**渲染模式** .

光源类型指的是该光源是平行光还是其他类型的光源，而光源的渲染模式指的是该光源是否是**重要的 (Important)** . 如果我们把一个光源的渲染模式设置为 Important，意味着我们告诉 Unity，该光源很重要，把它当成一个逐像素光源来处理 .

在前向渲染中，当我们渲染一个物体时，Unity 会根据场景中各个光源的设置以及这些光源对物体的影响程度 (距离该物体的远近，光源强度等) 对这些光源进行一个重要度排序 . 其中，一定数目的光源会按逐像素的方式处理，然后最多有 4 个光源按逐顶点的方式处理，剩下的光源可以按 SH 方式处理 . Unity 使用的判断规则如下 :

- 场景中最亮的平行光总是按逐像素处理的 .
- 渲染模式被设置成 **Not Important** 的光源，会按逐顶点或者 SH 处理 .
- 渲染模式被设置成 **Important** 的光源，会按逐像素处理 .

- 如果根据以上规则得到的逐像素光源数量小于 **Quality Setting** 中的逐像素光源数量，会有更多的光源以逐像素的方式进行渲染 .

那么，在哪里进行光照计算呢？当然是在 Pass 里 . 前向渲染有两种 Pass : **Base Pass** 和 **Additional Pass** . 通常来说，这两种 Pass 使用的标签和渲染设置以及常规光照计算如下图所示 .

内置的光照变量和函数

根据我们使用的渲染路径 (即 Pass 标签中 LightModel 的值) , Unity会把不同的光照变量传递给 Shader .

在 Unity 5 中，对于前向渲染 (即 **LightModel** 为 **ForwardBase** 或 **ForwardAdd**) 来说，下表给出了我们可以在 Shader 中访问到的光照变量 .

名称	类型	描述
_LightColor0	float4	该 Pass 处理的逐像素光源的颜色
_WorldSpaceLightPos0	float4	_WorldSpaceLightPos0.xyz 是该 Pass 处理的逐像素光源的位置 . 如果该光源是平行光，那么 _WorldSpaceLightPos0.w 是 0 , 其他光源类型 w 值为 1
_LightMatrix0	float4*4	从世界空间到光源空间的变换矩阵 . 可以用于采样 cookie 和光照衰减 (attenuation) 纹理
unity_4LightPosX0 , unity_4LightPosY0 , unity_4LightPosZ0	float4	仅用于 Base Pass . 前 4 个非重要的点光源在世界空间中的位置
unity_4LightAtten0	float4	仅用于 Base Pass . 存储了前 4 个非重要的点光源的衰减因子
unity_LightColor	half[4]	仅用于 Base Pass . 存储了前 4 个非重要的点光源的颜色

前向渲染中可以使用的内置光照函数如下表所示 :

函数名	描述
float3 WorldSpaceLightDir(float4 v)	仅用于前向渲染中 . 输入一个模型空间中的顶点位置，返回世界空间中从该点到光源的光照方向 . 内部实现使用了 UnityWorldSpaceLightDir 函数，没有被归一化

函数名	描述
float3 UnityWorldSpaceLightDir(float4 v)	仅可用于前向渲染中 . 输入一个世界空间中的顶点位置 , 返回世界空间中从该点到光源的光照方向 . 没有被归一化
float3 ObjSpaceLightDir(float4 v)	仅可用于前向渲染中 . 输入一个模型空间中的顶点位置 , 返回模型空间中从该点到光源的光照方向 . 没有被归一化
float3 Shade4PointLights(...)	仅可用于前向渲染中 . 计算四个点光源的光照 , 它的参数是已经打包进矢量的光照数据 , 通常就是 unity_4LightPosX0 , unity_4LightPosY0 , unity_4LightPosZ0 , unity_LightColor 和 unity_4LightAtten0 等 . 前向渲染通常会使用这个函数来计算逐顶点光照

顶点照明渲染路径

顶点照明渲染路径是对硬件配置要求最少 , 运算性能最高 , 但同时也是得到的效果最差的一种类型 . 它不支持那些逐像素才能得到的效果 , 例如阴影 , 法线映射 , 高精度的高光反射等 . 实际上 , 它仅仅是前向渲染路径的一个子集 . 也就是说 , 所有可以在顶点照明渲染路径中实现的功能都可以在前向渲染路径中完成 . 就如它的名字一样 , 顶点照明渲染路径只是使用了逐顶点的方式来计算光照 .

Unity 中的顶点照明渲染

顶点照明渲染路径通常在一个 Pass 中就可以完成对物体的渲染 . 在这个 Pass 中 , 我们会计算我们关心的所有光源对该物体的照明 , 并且这个计算是按逐顶点处理的 . 这是 Unity 中最快速的渲染路径 , 并且具有最广泛的硬件支持 (但是游戏机上并不支持这种路径) .

延迟渲染路径

前向渲染的问题是 : 当场景中包含大量实时光源时 , 前向渲染的性能会急剧下降 , 这是因为计算量增长迅速 . 例如 , 如果我们在场景的某一区域放置了多个光源 , 这些光源影响的区域互相重叠 , 那么为了得到最终的光照效果 , 我们就需要为该区域的每个物体执行多个 Pass 来计算不同光源对物体的光照结果 , 然后在颜色缓冲区中把这些结果混合起来得到最终的光照 . 然而 , 每执行一个 Pass 我们都需要重新渲染一遍物体 , 但很多计算实际上是重复的 .

延迟渲染是一种更古老的渲染方法 , 但由于上述前向渲染可能造成的瓶颈问题 , 近几年又流行起来 . 除了前向渲染中使用的颜色缓冲和深度缓冲外 , 延迟渲染还会利用额外的缓冲区 , 这些缓冲区也被统称为 G 缓冲 (G-buffer) . 其中 G 是英文 Geometry 的缩写 . G 缓冲区存储了我们所关心的表面 (通常指的是

离摄像机最近的表面) 的其他信息 , 例如该表面的法线 , 位置 , 用于光照计算的材质属性等 . 还是一种用空间换时间的处理策略 .

延迟渲染的原理

延迟渲染主要包含了两个 Pass . 在第一个 Pass 中 , 我们不进行任何光照计算 , 而是仅仅计算哪些片元是可见的 , 这主要是通过深度缓冲技术来实现 . 当法线一个片元是可见的 , 我们就把它的相关信息存储到 G 缓冲区中 . 然后 , 在第二个 Pass 中 , 我们利用 G 缓冲区的各个片元信息 , 例如表面法线 , 视角方向 , 漫反射系数等 , 进行真正的光照计算 .