# Unity Shader

## 渲染管线

**·** 主要任务：由一个三维场景出发、生成（或者渲染）一张二维图像；

即计算机从模型等三维物体的一系列的顶点数据、纹理等信息出发，把这些信息最终转换成 一张人眼可见的图像。

概念上，可以将渲染流水线分为 3 个阶段：

**·** 应用阶段（CPU上进行，主要输出为渲染图元，传递给GPU进行下一步操作，CPU操作有准备场景数据阶段、提高渲染性能（culling剔除）、设置模型渲染状态）

**·** 几何阶段（GPU上进行，接收渲染图元，进行逐顶点和逐多边形的操作，主要任务是顶点坐标变换到屏幕空间，再由光栅器处理；主要输出为输出屏幕空间的二维顶点坐标、每个顶点对应的深度值、着色等相关信息）

**·** 光栅化阶段（GPU上进行，主要任务为决定每个渲染图元中的那些像素应该被绘制在屏幕上，即接收几何阶段的输出数据，对这些逐顶点数据（如纹理坐标、顶点颜色等）进行插值变换，然后进行逐像素处理）

上述渲染流水线也分为 2 个阶段：

**·** CPU和GPU通信，起点为CPU

1、把数据加载到显存（硬盘——>RAM（系统内存）——>VRAM（显存））

说明：显卡一般没有对内存的访问权限，但对显存有较快的访问速度；从硬盘加载数据到RAM是比较耗时的操作

2、设置渲染状态

渲染状态：使用何种着色器（顶点着色器（Vertex Shader）、片元着色器（Fragment Shader）)、光照属性、材质等

3、调用DrawCall

DrawCall：一个命令，有CPU发起，GPU接收该命令，GPU可以开始一个渲染过程，即进行下一阶段的GPU渲染流水线

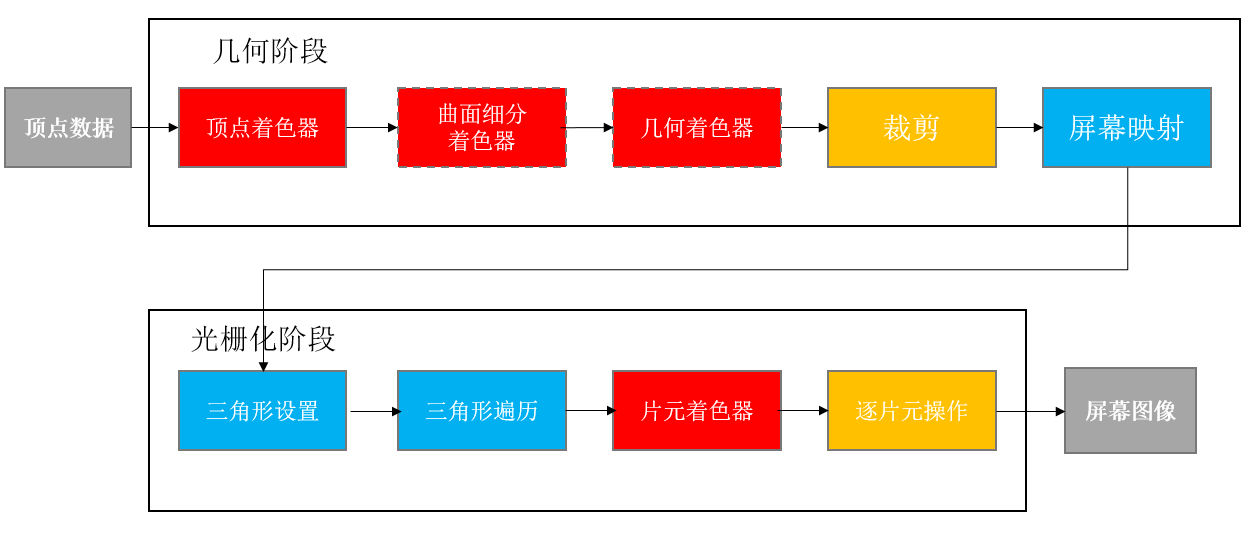
注意：该命令仅指向一个需要被渲染的图元列表，不包括任何材质信息；

### GPU渲染管线

**·** GPU流水线（又称渲染流水线或者渲染管线，**不同于上述概念性渲染流水线**）

当GPU从CPU那得到渲染命令后，就会进行一系列流水线操作，最终把图元渲染到屏幕上；

这些流水线操作开发者无法拥有绝对的控制权，但开放了很多控制权。



如上图所示：颜色表示了不同阶段的可配置性或可编程性：

红色表示该流水线阶段是完全可编程控制的，

黄色表示该流水线阶段可以配置但不是可编程的，

蓝色表示该流水线阶段是由GPU固定实现的，开发者没有任何控制权限。

**实现**表示该Shader必须由开发者编程实现，

虚线表示该Shader是可选的，

一般讲片元着色器默认为编程实现。

#### 顶点着色器

**高度可编程（详见后面叙述的三种着色器）**

顶点着色器是渲染管线的第一个阶段，从CPU获取输入，处理单位为顶点，即输入进来的每个顶点都可以调用一次顶点着色器；

顶点着色器本身不能创建或者销毁顶点，而且也无法得到顶点与顶点间的关系，如无法得知两个顶点是否属于同一三角网格；由于该相互独立的特性，GPU可以利用本身的特性并行化处理每个顶点，大幅度提高处理速度。

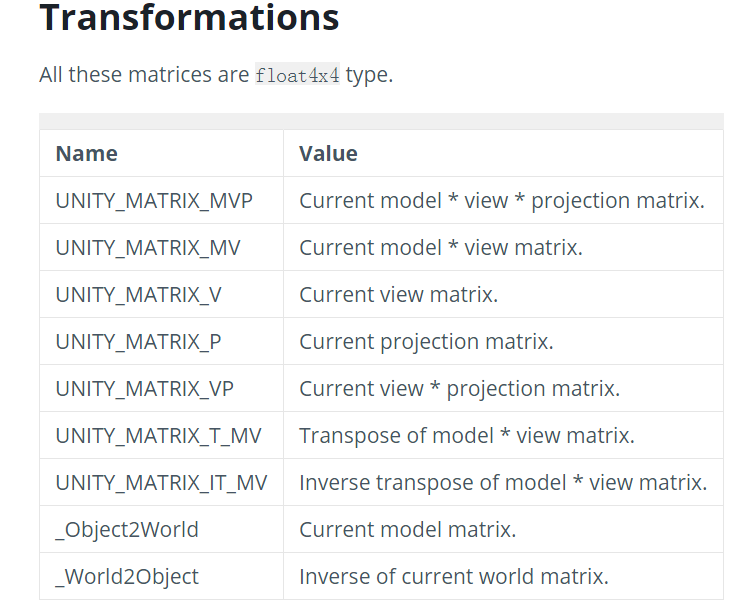
顶点着色器主要需要完成工作有：***坐标变换***（矩阵变换：3D数学待补充）和逐顶点光照。

**坐标变换**：对顶点坐标进行相应的坐标系转换（模型空间——>世界空间——>观察空间（相机空间）——>齐次裁剪空间（为裁剪做准备）），主要通过与**变换矩阵**相乘实现（Unity内置了一些变换矩阵：如UNITY\_MATRIX\_MVP（当前的模型·观察·投影矩阵，用于将顶点/方向矢量从模型空间变换到裁剪空间），使用方式如：

“o.pos = mul(UNITY\_MATRIX\_MVP , v.vertex )”）

更多***内置变换及其他内置属性*** 请参见：

<https://docs.unity3d.com/Manual/SL-UnityShaderVariables.html>



逐顶点光照：（后续标准光照模型还会涉及）

可以在顶点着色器中使用逐顶点光照方式实现标准光照模型（自反光、高光反射、漫反射、环境光）；同样的在接下来片元着色器中也有逐像素光照方法来实现光照模型

可以在顶点着色器中修改顶点颜色，顶点位置变换并输出给后续阶段

#### 曲面细分着色器（待深入）

细分图元

#### 几何着色器（待深入）

用于执行逐图元的着色操作，或者被用于产生更多的图元

#### 裁剪

裁剪掉不在摄像机视野内的顶点，并剔除默写三角图元的面片；

该阶段是可配置的（可配置见后面叙述的三种着色器），如可以自定义裁剪平面来配置裁剪区域，可以通过指令控制裁剪的三角图元的正面还是背面等

#### 屏幕映射

不可编程和配置，负责把每个图元的坐标（x和y）转换到屏幕（屏幕坐标系）上；

#### 三角形设置

**固定函数阶段**

开始进入光栅化(把顶点数据转换为片元的过程)阶段，该阶段会计算光栅化一个三角网格所需的信息，

为了得到每个三角网格对像素的覆盖情况，必须计算每条边上的像素坐标，此时就需要得到三角形边界的表示方式 =====》 这样一个计算三角网格表示数据的过程就叫做三角形设置，并输出

#### 三角形遍历

**固定函数阶段**

检查每个像素是否被一个三角网格所覆盖，若覆盖，则生成一个对应的片元，这个过程就是三角形遍历

注意：片元不仅仅是像素，而是包含了很多状态的集合，用于计算每个像素的最终颜色；如屏幕坐标、深度信息，以及其他从几何阶段输出的顶点信息（法线和纹理坐标等）

#### 片元着色器

**高度可编程的（详见后面叙述的三种着色器）**

**输入 上一阶段对顶点信息插值（逼近取近似值）得到的结果；**

插值：用来填充[图像变换](http://baike.baidu.com/view/2844853.htm)时像素之间的空隙。

主要实现：（shader 中 默认定义为 texcoord0/1/2/3... 参数）纹理采样：获取顶点着色器中输出的顶点对应的纹理坐标，该坐标再经过光栅化对三角网格的3个顶点对应的纹理坐标进行插值，最终得到其覆盖像素的片元的纹理坐标

注意：片元着色器仅影响单个片元

对应于顶点着色器，此时片元着色器可编写逐像素光照实现光照模型（后续标准光照模型还会涉及）

#### 逐片元操作

**高度可配置的，无法修改底层，可控制开启关闭功能**

主要任务：1、决定每个片元的可见性，包含所有**测试工作**，例如深度测试、模板测试等**（详见后面叙述的三种着色器）**

2、如果一个片元通过了所有的测试，就需要把这个片元的颜色值和已经存储在颜色缓冲区中的颜色进行合并，或者说混合

简单来说：

片元一般经过 模板测试（通常用于限制渲染区域，和渲染阴影、轮廓渲染等） 将通过测试的片元储存到模板缓冲区，下一步进行深度测试（把当前片元与深度缓冲区中的片元进行深度值得比较），接下来可以进行合并操作（混合操作，涉及到透明物体和不透明物体的颜色值合并（如对透明通道的相加、相减、相乘等等）），最终得到混合后的图像

**注意：**针对半透明物体的性能下降原因：现在GPU已经是现在将深度测试放在片元着色器之前（称为 Early-Z）,从而更早的剔除无法通过深度测试的片元以节省渲染时间；

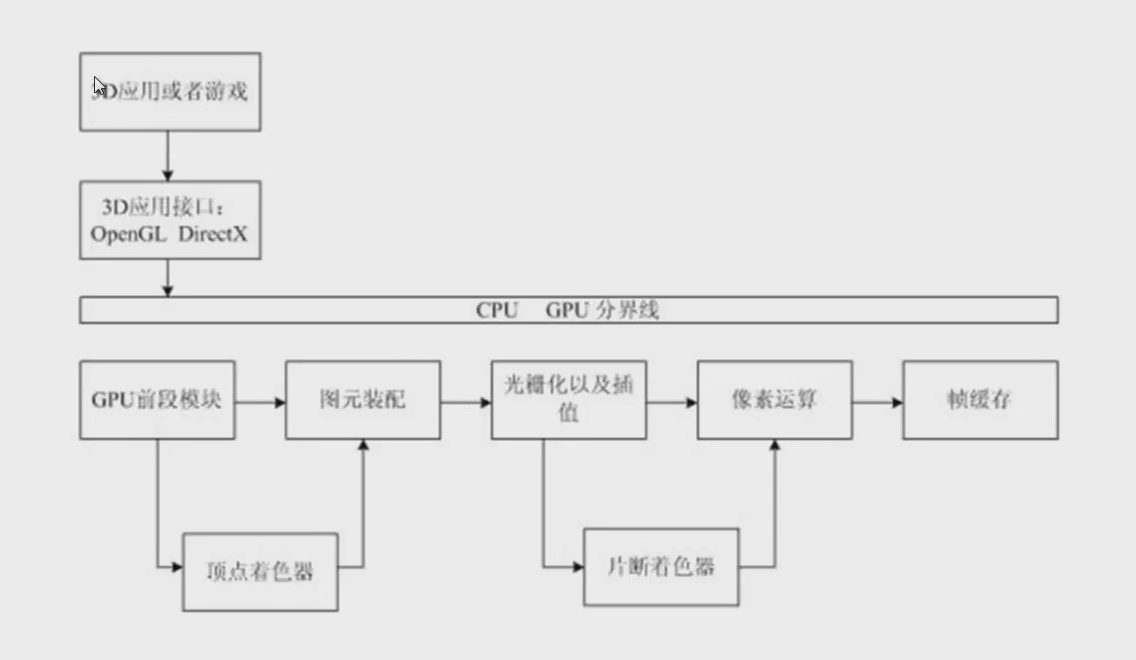
然而由于透明物体需要开启透明度测试以达到实际透明效果，需要在片元着色器中对透明度进行测试（透明度测试），无法提前进行深度测试，从而导致渲染性能的下降

#### 总结

**上述GPU渲染管线/流水线各阶段，在Unity中封装了很多功能，更多时候我们只需要为一个shader设置一些输入，编写顶点着色器和片元着色器，或者是Unity的表面着色器（核心还是顶点和片元着色器的变形），就能实现效果**

## OpenGL、DirectX是啥

均为图像应用程序的接口，沟通上层应用程序和GPU的传递渲染指令。



### 坐标系不同

OpenGL：以左下角为原点，y轴向上为正，x轴向右为正

DirectX： 以左上角为原点，y轴向下为正，x轴向右为正

注意：Unity为OpenGL坐标系，unity shader编程从CPU传入的数据如果转到DirectX下，需要进行坐标系的转换

当在Unity中进行屏幕后期处理时，如果开启了抗锯齿（Edit-PS-Quality-Anti Aliasing）,

在DirectX下，得到经过屏幕处理后的图像会被Unity自动调整（翻转），图像朝向是正确的，但是如果要同时渲染多张图像（包含主纹理和其他纹理），并开启抗锯齿，需要对其他纹理进行采样坐标的竖直方向的翻转。

### CG语言

三大主流的GPU编程语言（shader language）：HLSL 、 GLSL 、 CG

GLSL：基于OpenGL的OpenGL Shading Language

HLSL：基于DirectX的High Level Shading Language

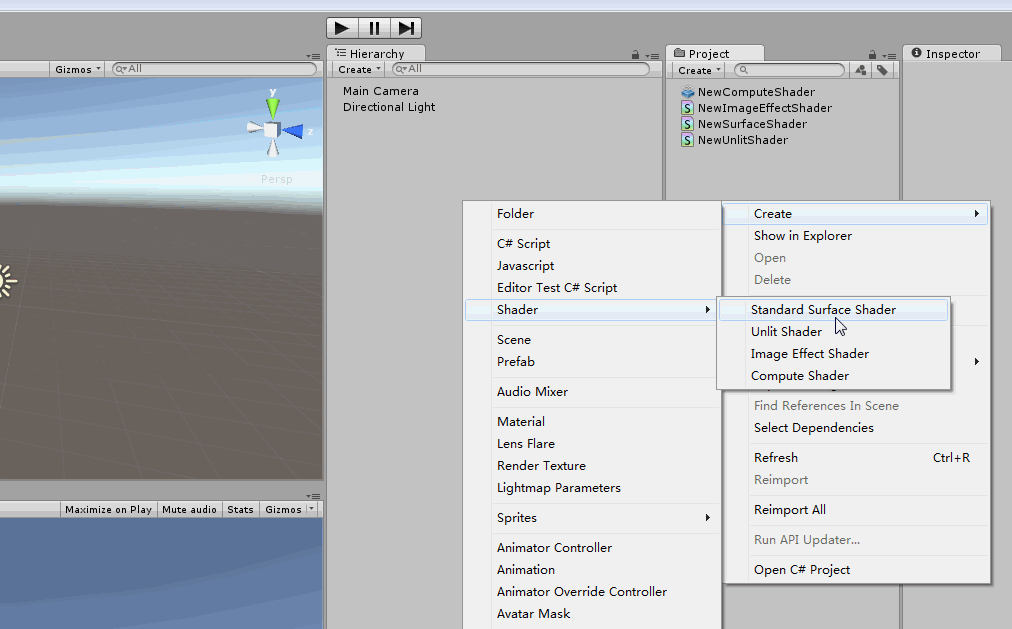
CG：Nvidia 的 C for Graphic

**在Unity中的shaderLab（就是unity shader 语言库）** ：

主要使用的是 Cg/HLSL 的代码片段 ，也支持GLSL（但推荐使用原生的GLSL）

## Unity Shader形式（三类着色器编写）

1、Unity Shader的创建



如上图所示，

其中：Standard Surface Shader 为一个包含了标准光照模型（的表面着色器模板；

    Unlit Shader 则会产生一个不包含光照（但包含雾效）的基本的顶点（Vertex）/片元（Fragmaent）着色器；

    Image Effect Shader 提供了实现各种屏幕后期处理效果的一个基本模板；

    Compute Shader 为一种旨在利用GPU的并行性来进行一些与常规渲染流水线无关的计算 的一种特殊Shader。

2、Unity Shader 之 ShaderLab

首先，ShaderLab时Unity提供的编写Unity Shader的一种说明性语言，Unity Shader都是基于ShaderLab编写的。

**部分关键词可以不区分大小写，如Pass,subshader等**

ShaderLab基础结构如下***（下列三种着色器不一定包括下面所有的结构）***：

Shader "ShaderName"//**常用格式为 "Custom/MyShader"，也是该Shader所在位置**

{

  Properties

{

     //属性

     //Properties块内的语法是单行的

}

SubShader//可多个，设置为针对不同显卡

{

Tags{//标签设置}

XXX //渲染状态配置

Pass{//Pass通道，可多个}

}

FallBack “//回滚到一个系统内置的shader”

}

其中：

**Properties形式如下：**



 说明：

**·**内部名称（name索引），在Unity中一般以“\_”开头命名

**·**如下的一些形式中 在“=”后面的“   ”中的值可以缺省

**·**Range和float形式的属性中 只能为 单精度值

**·**对于纹理(2D, Rect, Cube) 缺省值既可以是一个空字符串也可以是某个内置的缺省纹理："white", "black", "gray" or"bump"

   实现声明，方便识别

   white black red  gray =>2D Texture /  Detail Texture

   bump => BumpMap（凹凸贴图）

**注意：之后的调用属性值 需要在subshader中定义时需要通过完整的【Name】名来访问，有下划线也要保留**

 properties其他形式有：

**name *("display name", Range (min, max)) =number***

        定义浮点数属性，在检视器中可通过一个标注最大最小值的滑条来修改。

**name *("display name", Color) =(number,number,number,number)***

        定义颜色属性

**name *("display name", 2D) = "name" {options }***

      定义2D纹理属性

**{option}的属性可选：**

**TexGen纹理生成类型**：即纹理的自动生成纹理坐标时的模式，可以是ObjectLinear, EyeLinear,SphereMap, CubeReflect, CubeNormal的其中之一;这些模式和OpenGL纹理生成模式相对应。注意如果使用自定义顶点程序（顶点着色器），那么纹理生成将被忽略。

**LightmapMode光照贴图模式**：如果我们给出这个选项，纹理将能被渲染器的光线贴图属性所影响。纹理不能被使用在材质中，而是取自渲染器的设定。

**name *("display name", Rect) = "name"{ options }***

  定义长方形（非2次方）纹理属性

**name *("display name", Cube) = "name"{ options }***

  定义立方贴图纹理属性

**name *("display name", Float) = number***

  定义浮点数属性

**name *("display name", Vector) =(number,number,number,number)***

  定义一个四元素的容器(相当于Vector4)属性

3、子着色器（Subshader）

**Subshader*{ [Tags] [CommonState] Passdef [Passdef ...] }***

3.1 SubShader子着色器标签（SubShader Tags）

标签作用：

通知渲染引擎期望何时与如何渲染对象

标签语法：（多组键值对）

**Tags*{ "TagName1" ="Value1" "TagName2" = "Value2" }***

3.1.1 队列标签（Queue Tag）: 决定渲染次序

<http://blog.csdn.net/candycat1992/article/details/37345251>



常用技巧：中间队列（简单地说，就是自定义渲染队列值）

像透明的水，我们总希望应该在所有不透明物体之后并且在透明对象前被渲染，可以选择中间队列（***Tags{"Queue"="Geometry + 1"}***）

3.1.1.1 着色器分类标签（RenderType）

***Tags{"RenderType"="Transparent"}***

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: Skybox shaders. 天空盒着色器。  
Overlay: GUITexture, Halo, Flare shaders. 光晕着色器、闪光着色器。  
TreeOpaque: terrain engine tree bark. 地形引擎中的树皮。  
TreeTransparentCutout: terrain engine tree leaves. 地形引擎中的树叶。  
TreeBillboard: terrain engine billboarded trees. 地形引擎中的广告牌树。  
Grass: terrain engine grass. 地形引擎中的草。  
GrassBillboard: terrain engine billboarded grass. 地形引擎何中的广告牌草。

3.1.2 忽略投影标签

***Tags{"IgnoreProjector"="true"}***对于半透明物体，由于对半透明物体产生投影目前暂时还没有比较合适的方法，可以直接使用该标签进行忽略。

3.1.3 禁用批处理标签

***Tags{"DisableBatching"="true"}***一些如使用了模型空间下的坐标进行顶点动画的着色器会在Unity的批处理时出问题，可以直接禁用对该shader禁用批处理

**渲染路径（光照模型时进行选择）**

***Tags{"LightMode"="ForwardBase"}***

* **Always**:     不管使用那种渲染路径，该Pass总会被渲染，但不会计算任何光照
* **ForwardBase**:  用于“前向渲染（Forward Rendering）”，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（shadermap）或一张深度纹理
* **PrepassBase**:  用于“遗留的延迟渲染”，该Pass会渲染法线和高光反射的指数部分
* **PrepassFinal**: 用于“遗留的延迟渲染”，该Pass通过合并纹理、光照和自发光来渲染最后得到颜色

**用于遗留的顶点照明渲染**

* Vertex:  用于顶点光照渲染，当物体没有光照映射时，应用所有的顶点光照
* VertexLMRGBM:用于顶点光照渲染，当物体有光照映射的时候使用顶点光照渲染。在平台上光照映射是RGBM 编码
* VertexLM: 用于顶点光照渲染，当物体有光照映射的时候使用顶点光照渲染。在平台上光照映射是double-LDR 编码（移动平台，及老式台式CPU）

**3.1.4 RenderSetup 设置渲染状态**

注意：在Subshader上设置渲染状态时，将会运用到所有Pass上，可以在每个Pass上设置单独的渲染状态；

通道设定显示硬件的各种状态，例如能打开alpha混合，能使用雾，等等。这些命令如下：

**Material *{ Material Block }***

定义一个使用顶点光照管线的材质,详情参考上次我们讲的Material

**Lighting *On | Off***

开启或关闭顶点光照。开启灯光之后，顶点光照才会有作用

**Cull *Back | Front | Off***

设置多边形剔除模式，详细内容后面的文章会讲解到。

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**ZTest*(Less | Greater | LEqual | GEqual |Equal | NotEqual | Always | Off | Never)***

设置深度测试模式，默认值为 LEqual，代表如何将像素的颜色写入深度缓存中，

例如，当去默认值LEqual时，如果将要绘制的新像素的z值<= 深度缓存中的值，则将新像素的

 颜色值更新深度缓存中对应像素的颜色值

**需要注意的是，当ZTest取值为Off时，表示的是关闭深度测试，等价于取值为Always，而不是Never！Always指的是直接将当前像素颜色(不是深度)写进颜色缓冲区中；而Never指的是不要将当前像素颜色写进颜色缓冲区中，相当于消失。**

具体测试结果参考：<http://www.unity.5helpyou.com/3090.html>

**ZWrite *On | Off***

设置深度写模式，默认值为On，代表如何将像素的深度写入深度缓存中，

（1）什么是深度？

深度其实就是该像素点在3d世界中距离摄象机的距离，深度值Zbuffer（Z值）越大，则离摄像机越远。

（2）什么是深度缓存？

深度缓存中存储着每个像素点（绘制在屏幕上的）的深度值！如果启用了深度缓冲区，在绘制每个像素之前，OpenGL会把它的深度值和已经存储在这个像素的深度值进行比较。新像素深度值<原先像素深度值，则新像素值会取代原先的；反之，新像素值被遮挡，其颜色值和深度将被丢弃，最终屏幕显示的就是深度缓存中深度对应的像素点的颜色！(深度主要起的是比较的作用)

（3）什么是深度测试？

在深度测试中，默认情况是将要绘制的新像素的z值与深度缓冲区中对应位置的z值进行比较，如果比深度缓存中的值小，那么用新像素的颜色值更新深度缓存中对应像素的颜色值。

（4）为什么需要深度？

在不使用深度测试的时候，如果我们先绘制一个距离较近的物体，再绘制距离较远的物体，则距离远的物体因为后绘制，会把距离近的物体覆盖掉，这样的效果并不是我们所希望的。而有了深度缓冲以后，绘制物体的顺序就不那么重要了，都能按照远近（Z值）正常显示，这很关键。(越后绘制的东西，距离相机就越近)

在Unity中，先根据渲染队列（Queue）中队列值较小的先进行渲染，再执行Zwrite \ ZTest

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**Fog *{ Fog Block }***

设置雾参数，

**AlphaTest *(Less | Greater | LEqual | GEqual| Equal | NotEqual | Always) CutoffValue***

开启alpha测试

**Blend *SourceBlendMode |DestBlendMode***

设置alpha混合模式

**Color *Color value***

设置当顶点光照关闭时所使用的颜色

**ColorMask *RGB | A | 0 | any combination of R, G, B, A***

设置颜色写遮罩。设置为0将关闭所有颜色通道的渲染

**Offset*OffsetFactor , OffsetUnits***

设置深度偏移

**SeparateSpecular *On | Off***

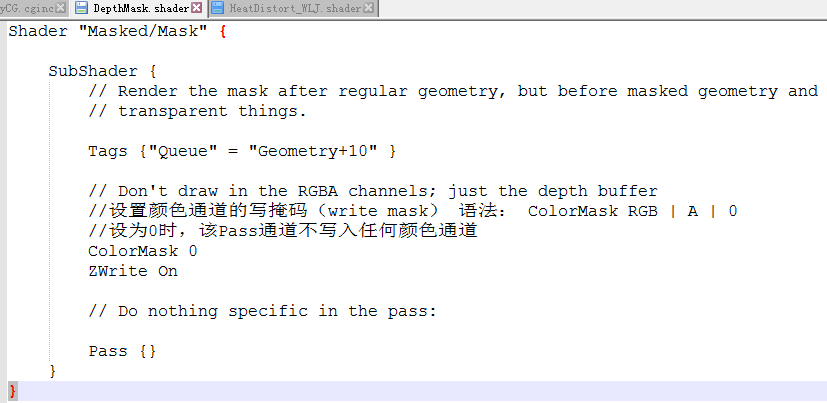
开启或关闭顶点光照相关的平行高光颜色。

**ColorMaterial *AmbientAndDiffuse | Emission***

当计算顶点光照时使用每顶点的颜色

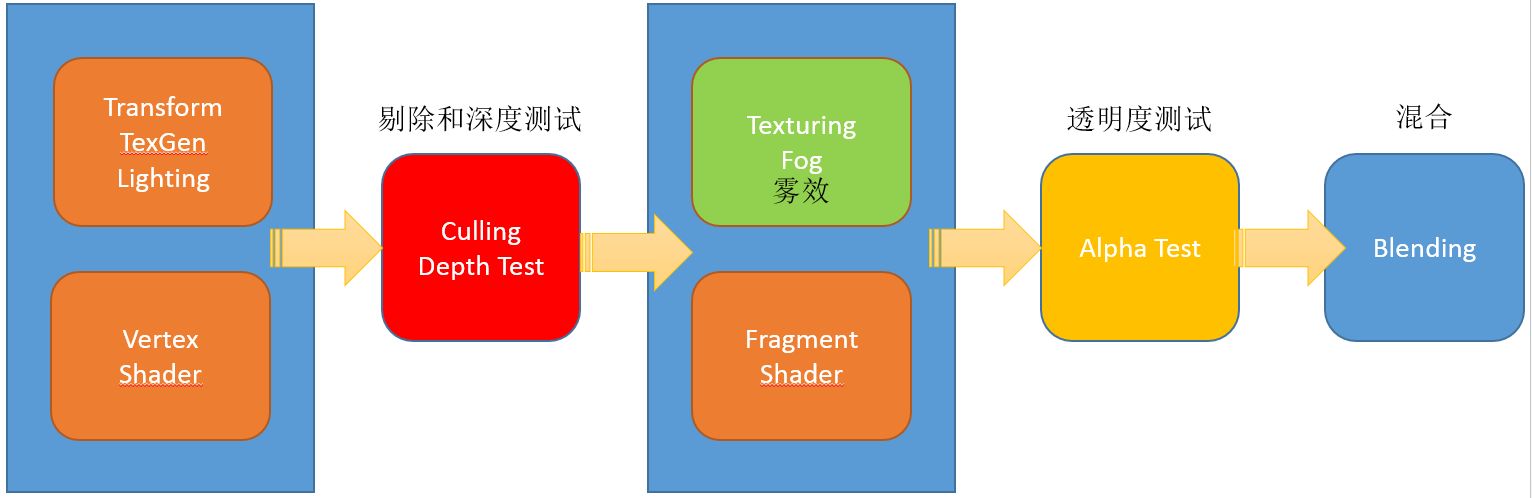
应用：

使用了Tags{“Queue”=“Geometry+10”}，表明队列在不透明通道前面



剔除、测试和混合进一步解释：

### 渲染管线中Shader渲染的大致流程



**· 剔除（Culling）**：将被挡住或者视野以外的不可见的物体，根据优化需求，可以选择不绘制，以节省资源。

**· 深度测试（Depth Testing）**: 多个物体的遮挡，离相机的Z轴方向的距离大小

### 剔除和深度测试的一些关键词：

#### · Cull Back / Front / Off

* Cull Back—— 不绘制背离观察者的几何面
* Cull Front—— 不绘制面向观察者的几何面，用于由内自外的旋转对象
* Cull Off —— 显示所有面。用于特殊效果。

#### · ZWrite On /Off

**(是否和深度缓冲区内的深度值做比较)**

       用于控制是否将来之对象的像素写入深度缓冲（默认开启），

       如果需要绘制纯色物体，便将此项打开，也就是写上ZWrite On。

       如果要绘制半透明效果，关闭深度缓冲，则用ZWrite Off。

#### · ZTest Less/ Greater / LEqual / GEqual / Equal / NotEqual /Always

       用于控制深度测试如何执行。

        缺省值是LEqual （绘制和存在的对象一致或是在其中的对象；隐藏其背后的对象）

|  |  |
| --- | --- |
| Greater | 只渲染大于AlphaValue值的像素 |
| GEqual | 只渲染大于等于AlphaValue值的像素 |
| Less | 只渲染小于AlphaValue值的像素 |
| LEqual | 只渲染小于等于AlphaValue值的像素 |
| Equal | 只渲染等于AlphaValue值的像素 |
| NotEqual | 只渲染不等于AlphaValue值的像素 |
| Always | 渲染所有像素，等于关闭透明度测试。等于用AlphaTest Off |
| Never | 不渲染任何像素 |

#### · Offset Factor , Units

此语句用两个参数（Facto和Units）来定义深度偏移。

* Factor参数表示 Z缩放的最大斜率的值。
* Units参数表示可分辨的最小深度缓冲区的值。

于是，我们就可以强制使位于同一位置上的两个集合体中的一个几何体绘制在另一个的上层。例如偏移量Offset 设为0, -1（即Factor=0, Units=-1）的值使得靠近摄像机的几何体忽略几何体的斜率，而偏移量为-1,-1（即Factor =-1, Units=-1）时，则会让几何体偏移一个微小的角度，让观察使看起来更近些。

**· 透明度测试（Alpha Testing）**: 在unity中是阻止像素被写到屏幕的最后机会；

       在片元计算出最终需要渲染的颜色之后，可以通过将颜色的透明度和一个固定值进行比较，如果比较失败，则该像素不会显示输出。

      Alpha测试在渲染凹形物体的透明部分时非常有用。显卡上有着每个像素写到屏幕上的深度记录。如果一个新的像素比原来的像素的深度深，那么新的像素就不会被写到屏幕中。

**透明度测试的一些关键词：**

#### · AlphaTest Off用于渲染所有像素（默认缺省）

**· AlphaTest comparison AlphaValue**

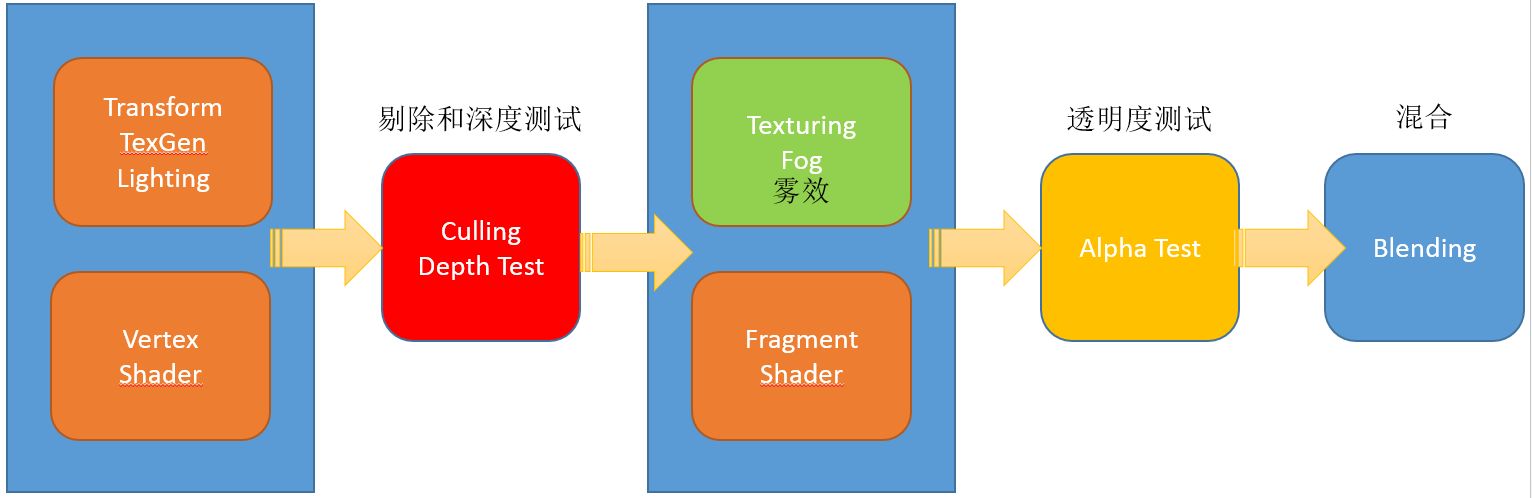
   用于设定透明度测试只渲染在某一确定范围内的透明度值的像素。其中的comparison取值为下表之一：

   而AlphaValue为一个范围在0到1之间的浮点值。也可以是一个指向浮点属性或是一个范围属性，在后一种情况下需要使用标准的方括号写法标注索引名字，如(   **[变量名]**).

|  |  |
| --- | --- |
| **Greater** | Only render pixels whose alpha is greater than AlphaValue. 大于 |
| **GEqual** | Only render pixels whose alpha is greater than or equal to AlphaValue. 大于等于 |
| **Less** | Only render pixels whose alpha value is less than AlphaValue. 小于 |
| **LEqual** | Only render pixels whose alpha value is less than or equal to from AlphaValue. 小于等于 |
| **Equal** | Only render pixels whose alpha value equals AlphaValue. 等于 |
| **NotEqual** | Only render pixels whose alpha value differs from AlphaValue. 不等于 |
| **Always** | Render all pixels. This is functionally equivalent to AlphaTest Off.  渲染所有像素，**等于关闭透明度测试AlphaTest Off** |
| **Never** | Don't render any pixels. **不渲染任何像素** |

**· 混合（Blend）: Shader渲染的最后一步**

**常用于制作透明物体、进行纹理混合**

****

如上图所示，正被渲染的像素经过顶点光照、顶点着色器、剔除和深度测试，雾效、Alpha测试等一系列操作之后，最后一步便是混合操作。这个时候计算结果即将被输出到帧缓冲中。而混合操作，就是管理如何将这些像素输出到帧缓存中的这样一个过程——是直接替换原来的，是一加一的混合，还是有Alpha参与的不等比地混合等等。

混合操作有两个对象：源和目标，因此也有两个对应的因子，即源因子和目标因子（对应于Blend SrcFactor DstFactor操作）。

而如果我们把RGB颜色通道和Alpha通道分开来操作的话，混合就有了4个操作对象（对应于Blend SrcFactor DstFactor,SrcFactorA DstFactorA操作）。

### 混合操作的关键词：

#### Blend Off

Turn off blending 关闭混合

#### Blend SrcFactor DstFactor

基本的配置并启动混操作。

对产生的颜色乘以SrcFactor.对 已存在于屏幕的颜色乘以DstFactor，并且两者将被叠加在一起。

#### Blend SrcFactor DstFactor, SrcFactorA DstFactorA

同上，但是使用不同的要素来混合alpha通道，也就是有了4个操作对象

#### BlendOp Add /Min | Max | Sub | RevSub

此操作不和Blend操作一样  添加混合颜色在一起  ，而是   对它们做不同的操作  。

而如下便是常用混合操作符（blend operations）的含义列举：

|  |  |
| --- | --- |
| **Add** | 将源像素和目标像素相加. |
| **Sub** | 用源像素减去目标像素 |
| **RevSub** | 用目标像素减去源像素 |
| **Min** | 取目标像素和源像素颜色的较小者作为结果 |
| **Max** | 取目标像素和源像素颜色的较大者作为结果 |

**混合因子（Blend factors）:**

以下所有的属性都可作为**SrcFactor**或**DstFactor**。其中，Source指的是被计算过的颜色，Destination是已经在屏幕上的颜色。

|  |  |
| --- | --- |
| **One** | 值为1，使用此因子来让帧缓冲区源颜色或是目标颜色完全的通过。 |
| **Zero** | 值为0，使用此因子来删除帧缓冲区源颜色或目标颜色的值。 |
| **SrcColor** | 使用此因子为将当前值乘以帧缓冲区源颜色的值 |
| **SrcAlpha** | 使用此因子为将当前值乘以帧缓冲区源颜色Alpha的值。 |
| **DstColor** | 使用此因子为将当前值乘以帧缓冲区源颜色的值。（修改为屏幕上的颜色） |
| **DstAlpha** | 使用此因子为将当前值乘以帧缓冲区源颜色Alpha分量的值。（修改为屏幕上的颜色Alpha分量） |
| **OneMinusSrcColor** | 使用此因子为将当前值乘以(1 -帧缓冲区源颜色值) |
| **OneMinusSrcAlpha** | 使用此因子为将当前值乘以(1 -帧缓冲区源颜色Alpha分量的值) |
| **OneMinusDstColor** | 使用此因子为将当前值乘以(1 –目标颜色值) |
| **OneMinusDstAlpha** | 使用此因子为将当前值乘以(1 –目标Alpha分量的值) |

在Pass{}中使用如下：

1. Blend SrcAlpha OneMinusSrcAlpha    // Alpha混合
2. Blend One One                       // 相加
3. Blend One OneMinusDstColor         // 比较柔和的相加（SoftAdditive）
4. Blend DstColor Zero                  // 乘法
5. Blend DstColor SrcColor              // 2倍乘法

### 基本雾效（Fog）

雾效（Fog）参数用于控制雾相关的渲染

在计算机图形学中，雾化是通过混合已生成的像素的颜色和基于到镜头的距离来确定的最终的颜色来完成的。雾化不会改变已经混合的像素的透明度值，只是改变RGB值。

Unity中的缺省雾设定是基于Edit->RenderSettings中的设置的，且雾模式既可以是Exp2也可以是关闭；密度和颜色完全取自设定。

注意如果使用了片段着色器的话，着色器中的雾设定仍然是有效的。如果平台没有对固定管线的雾功能支持，Unity会实时补上着色器以支持需要的雾模式。

#### 雾效的一些关键词：

##### · Fog { *FogCommands* }

  用于设定雾命令的内容

***FogCommands* :**

**· Mode *Off / Global / Linear / Exp / Exp2***

用于定义雾模式。缺省是全局的，依据雾在渲染设定中是否打开确定可从无变化到平方值

**· Color *ColorValue***

    用于设定雾的颜色值

**· Density *FloatValue***

以指数的方式设定雾的密度

**· Range *FloatValue,FloatValue***

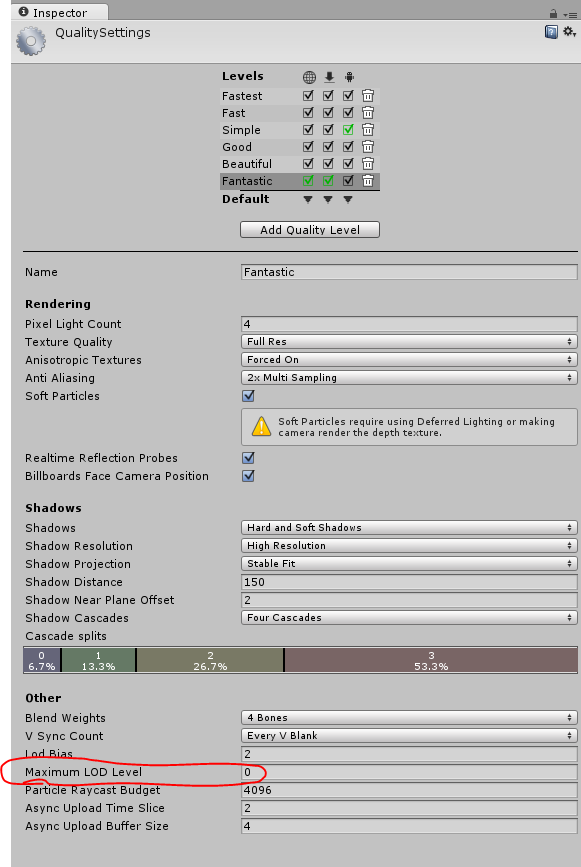
用于为linear类型的雾设定远近距离值

### LOD:

细节层(Level of Detail)

这个数值决定了我们能用什么样的Shader。在Unity的Quality Settings中我们可以设定允许的最大LOD，当设定的LOD小于SubShader所指定的LOD时，这个SubShader将不可用。Unity内建Shader定义了一组LOD的数值，我们在实现自己的Shader的时候可以将其作为参考来设定自己的LOD数值，这样在之后调整根据设备图形性能来调整画质时可以进行比较精确的控制。

* VertexLit及其系列 = 100
* Decal, Reflective VertexLit = 150
* Diffuse = 200
* Diffuse Detail, Reflective Bumped Unlit, Reflective Bumped VertexLit = 250
* Bumped, Specular = 300
* Bumped Specular = 400
* Parallax = 500
* Parallax Specular = 600



### Unity支持的Shader Target

Shader Model:微软剔除的一套规范，决定了shader的各个特性的能力强弱，（可使用的运算指令数目、寄存器个数等），等级越高，Shader的性能越强

#pragma target 2.0 默认Shader Target等级，相当于 Direct3D 9的shader model 2.0

#pragma target 3.0 常用

#pragma target 4.0 / 5.0 相当于Direct3D 10 / 11的Shader Model 4.0 / 5.0（目前仅在Direct11和XboxOne/PS4上支持）

### 固定管线着色器

**就FixedFunction Shader而言：**

只需要使用一些关键字按照一定编写规范，就能实现很多效果，但相较于可编程着色器，可实现功能较少；其在Unity底层进行了大量工作。其主要目的是为了兼容一些老式显卡，主要实现方式为**顶点光照**，其核心模块是位于SubShader{}下的Material{}块中，*注意一些效果实现（如玻璃，植被）*

固定管线着色器，在OpenGL2.0（2004年发布）仅支持可编程管线的着色器，Unity目前暂时还支持。

1. Shader "浅墨Shader编程/Volume5/固定功能的Shader示例"
2. {
3. //-------------------------------【属性】-----------------------------------------
4. Properties
5. {
6. \_Color ("主颜色", Color) = (1,1,1,0)
7. \_SpecColor ("高光颜色", Color) = (1,1,1,1)
8. \_Emission ("自发光颜色", Color) = (0,0,0,0)
9. \_Shininess ("光泽度", Range (0.01, 1)) = 0.7
10. \_MainTex ("基本纹理", 2D) = "white" {}
11. }
13. //--------------------------------【子着色器】--------------------------------
14. SubShader
15. {
16. //----------------通道---------------
17. Pass
18. {
19. //-----------材质------------
20. **Material**
21. {
22. //可调节的漫反射光和环境光反射颜色
23. Diffuse [\_Color]
24. Ambient [\_Color]
25. //光泽度
26. Shininess [\_Shininess]
27. //高光颜色
28. Specular [\_SpecColor]
29. //自发光颜色
30. Emission [\_Emission]
31. }
32. //开启光照
33. Lighting On
34. //开启独立镜面反射
35. //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*让顶点光照和纹理颜色可以结合\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
36. SeparateSpecular On
37. //设置纹理并进行纹理混合
38. SetTexture [\_MainTex]
39. {
40. //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*2 倍 RGB 和 Alpha \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
41. Combine texture \* primary DOUBLE, texture \* primary
42. }     }  }  }

再提供一个固定管线着色器示例：

**植被效果：**

**使用两个Pass通道的原因：**

**渲染树和植物时，透明度测试会出现尖锐的边缘，解决方法之一：渲染对象两次（首次通道中只渲染超过50%透明度的像素，第二次通道中将透明度和上次剔除部分混合，且不记录像素深度（因为可能需要使一些树枝类的覆盖近的其他树枝））**

Shader "摘自浅墨博客"

{

//-------------------------------【属性】-------------------------------------

Properties

{

\_Color ("主颜色", Color) = (.5, .5, .5, .5)

\_MainTex ("基础纹理 (RGB)-透明度(A)", 2D) = "white" {}

\_Cutoff ("Alpha透明度阈值", Range (0,.9)) = .5

}

//--------------------------------【子着色器】--------------------------------

SubShader

{

//【1】定义材质

Material

{

Diffuse [\_Color]

Ambient [\_Color]

}

//【2】开启光照

Lighting On

//【3】关闭裁剪，渲染所有面，用于接下来渲染几何体的两面

Cull Off

//--------------------------【通道一】-------------------------------

// 说明：渲染所有超过[\_Cutoff] 不透明的像素

//----------------------------------------------------------------------

Pass

{

AlphaTest Greater [\_Cutoff]

SetTexture [\_MainTex] {

combine texture \* primary, texture

}}

//----------------------------【通道二】-----------------------------

// 说明：渲染半透明的细节

//----------------------------------------------------------------------

Pass

{

// 不写到深度缓冲中

ZWrite off

// 不写已经写过的像素

ZTest Less

// 深度测试中，只渲染小于或等于的像素值

**AlphaTest LEqual [\_Cutoff]**

// 设置透明度混合

Blend SrcAlpha OneMinusSrcAlpha

// 进行纹理混合

SetTexture [\_MainTex]

{

combine texture \* primary, texture

}}}}

#### 固定管线着色器的核心配置：

纹理设置（Texture Setup）

在完成渲染设定后，我们可以指定一定数量的纹理和当使用 SetTexture 命令时所采用的混合模式：

**SetTexture *[texture property]{ [Combineoptions] }***

纹理设置，用于配置固定函数多纹理管线，当自定义fragment shaders 被使用时，这个设置也就被忽略掉了。

##### 一些概念

每像素光照（Per-pixel Lighting）

每像素光照管线通过多次通道渲染对象来完成。Unity渲染对象一次来获取阴影色和任何顶点光照。然后再在额外的并行通道中渲染出每像素光照的效果。

每顶点光照（Per——vertex Lighting）

每顶点光照是标准的Direct3D/OpenGL光照模式，通过计算每个顶点的光照来完成。Lighting on命令开启光照。而我们知道，光照被材质块，颜色材质和平行高光等命令所影响。

3.2.5 UsePass & GrabPass

UsePass "Specular/Base"  使用内置的高光着色器中名叫“Base”的通道。

Pass通道中定义的Name "MyPassName" 就是给通道命名后方便调用Use。

GrabPass 抓取屏幕并将结果纯纯在一张纹理中，通常在靠后的通道中使用，这个纹理能被用于后续的通道中完成一些高级图像特效。

简单示例：

1. Shader "GrabPassInvert"
2. {
3. SubShader
4. {
5. //在所有不透明几何体之后绘制
6. Tags { "Queue" = "Transparent" }
8. //捕获对象后的屏幕到\_GrabTexture中
9. GrabPass { }
11. //用前面捕获的纹理渲染对象，并反相它的颜色
12. Pass
13. <span style="white-space:pre">  </span>{
14. SetTexture [\_GrabTexture] { combine one-texture }
15. }
16. }
17. }

##### 纹理（Texturing）

纹理在基本的顶点光照计算完成之后被应用，这也就是SetTexture 命令必须放置在通道的末尾的原因了。在着色器中通过SetTexture 命令来完成。

需要注意的是，SetTexture 命令在使用了片段着色器时不会生效；因为在片段着色器（fragment shader）下像素操作被完全描述在着色器中。

我们可以在一个通道中使用多个SetTexture命令，SetTexture的所有纹理都是按照代码顺序来添加的（类似PS的图层操作）

SetTexture语法：

**SetTexture *[TexturePropertyName] { TextureBlock }***

解释：分配一个纹理，其中TexturePropertyName必须为一个纹理，也就是在shader最开始的Properties中的属性。在TextrueBlock中设置如何应用纹理，即纹理块控制纹理如何被应用。

而在纹理块中能执行3种命令：合并操作，矩阵操作、与常量颜色进行混合操作。

纹理合并命令

<http://blog.csdn.net/poem_qianmo/article/details/41175585>

**combine *src1* \* *src2***

将源1和源2的元素相乘。结果会比单独输出任何一个都要暗

**combine *src1* + *src2***

将将源1和源2的元素相加。结果会比单独输出任何一个都要亮

**combine *src1* - *src2***

源1 减去 源2

**combine *src1* +- *src2***

先相加，然后减去0.5（也就是添加了一个符号）

**combine *src1* lerp (*src2*) *src3***

使用源2的透明度通道值在源3和源1中进行插值，注意插值是反向的：当透明度值是1是使用源1，透明度为0时使用源3

**combine *src1* \* *src2* + *src3***

源1和源2的透明度相乘，然后加上源3

**combine *src1* \* *src2* +- *src3***

源1和源2的透明度相乘，然后和源3做符号加

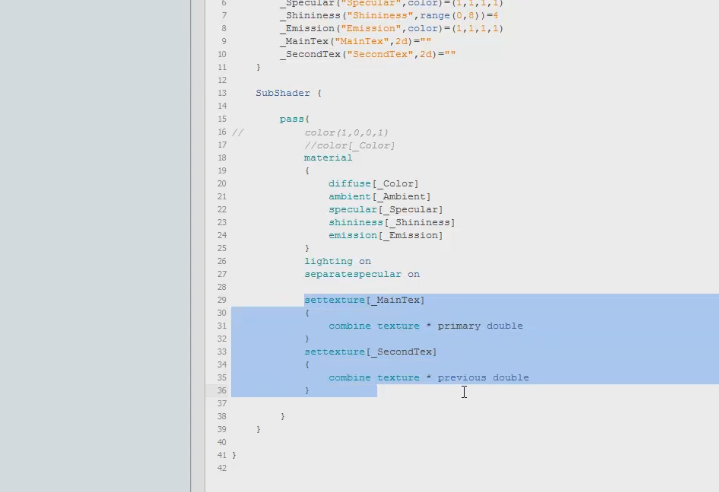
**combine *src1* \* *src2* - *src3***

源1和源2的透明度相乘，然后和源3相减

其中，所有src属性都可以是previous,constant, primary or texture其中的一个。

**Previous** 是上一次SetTexture的结果

**Primary** 是来自光照计算的颜色或是当它绑定时的顶点颜色

* 

**Texture是**在SetTexture中被定义的纹理的颜色

**Constant**是被ConstantColor定义的颜色

一些小技巧：

1.上述的公式都均能通过关键字 Double 或是 Quad 将最终颜色调高亮度2倍或4倍。

2.所有的src属性，除了差值参数都能被标记一个“-”负号来使最终颜色反相。

3.所有src属性能通过跟随 alpha 标签来表示只取用alpha通道。

3.3.2 颜色常量命令 & 纹理矩阵命令

**ConstantColor *color***

定义在combine命令中的能被使用的常量颜色

**matrix*[MatrixPropertyName]***

***使用给定矩阵变换纹理坐标***

3.3.3 分离的透明度和颜色混合（Separate Alpha & Color computation）

在默认情况下，混合公式被同时用于计算纹理的RGB通道和透明度。同时，我们也能指定针对透明度来单独计算，比如这样，将RGB操作和Alpha操作隔开：

**SetTexture*[\_MainTex] { combine previous \*texture, previous + texture }***

如上所述，我们对RGB的颜色做乘  然后 对Alpha透明度相加

### 表面（光照）着色器

***Surface Shader*：可以看做是一个表面光照Shader的语法块、一个光照顶点着色器和片元着色器的生成器；其中不包括Pass通道，其主要代码均包括在CGPROGRAM和ENDCG块中，（根据代码自动会编译成多个Pass{}）**

**严格上说，表面着色器本质是顶点着色器和片元着色器，所以本质来说，Unity Shader只有两种形式，顶点/片元着色器和固定功能（函数）着色器**

在表面着色器和顶点&片元着色器中都存在编译指令

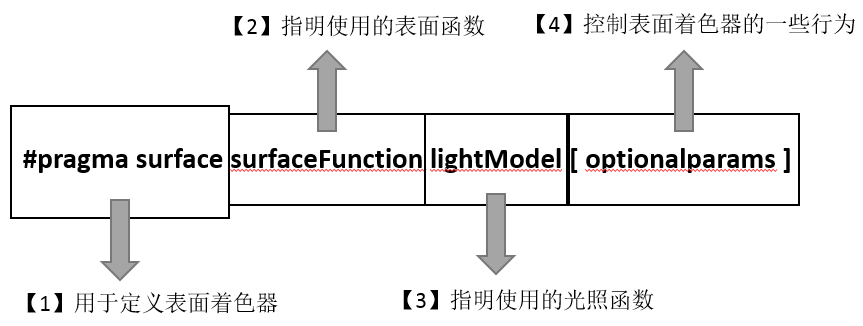
**说明：**

**表面着色器整体结构：**

**表面着色器放在CGPROGRAM...ENDCG块中，但必须放在SubShader块中，不是Pass{}里，区别于顶点/片元着色器的结构：SubShader{ CGPROGRAM{ Pass{ .. } }ENDCG }**

#### 编译指令：

一般在Surface Shader中的格式：



**【1】**：相类似的，顶点&片元着色器也有相应的函数定义，一般格式为：

#pragma vertex vert（vert即为定义的顶点着色器函数名）

#pragma fragment frag（frag即为定义的片元着色器函数名）

**【2】**：表面着色器相较于顶点&片元着色器，将“表面”这一概念抽离出来，“表面”包含了对象的反射率、光滑度、透明度等等值；

类似【1】中所说 通常将其命名成“surf”（当然也可以是其他的），在函数体重可以定义这些表面属性以实现不同效果，函数格式是固定的，如下：

void surf (**Input** IN, inout SurfaceOutput o) {　}

void surf (**Input** IN, inout SurfaceOutputStandard o) {　｝

void surf (**Input** IN, inout SurfaceOutputStandardSpecular o)　{　}

#### 两个结构体：

##### 输入 : Input

结构内容：

* float3 viewDir - 视图方向( view direction)值。为了计算视差效果(Parallax effects)，边缘光照(rim lighting)等，需要包含视图方向( view direction)值。
* float4 with COLOR semantic -每个顶点(per-vertex)颜色的插值。
* float4 screenPos - 屏幕空间中的位置。 为了反射效果，需要包含屏幕空间中的位置信息。比如在Dark Unity中所使用的 WetStreet着色器。
* float3 worldPos - 世界空间中的位置。
* float3 worldRefl - 世界空间中的反射向量。如果表面着色器(surface shader)不写入法线(o.Normal)参数，将包含这个参数。 请参考这个例子：Reflect-Diffuse 着色器。
* float3 worldNormal - 世界空间中的法线向量(normal vector)。如果表面着色器(surface shader)不写入法线(o.Normal)参数，将包含这个参数。
* float3 worldRefl; INTERNAL\_DATA - 世界空间中的反射向量。如果表面着色器(surface shader)不写入法线(o.Normal)参数，将包含这个参数。为了获得基于每个顶点法线贴图( per-pixel normal map)的反射向量(reflection vector)需要使用世界反射向量(WorldReflectionVector (IN, o.Normal))。请参考这个例子： Reflect-Bumped着色器。
* float3 worldNormal; INTERNAL\_DATA -世界空间中的法线向量(normal vector)。如果表面着色器(surface shader)不写入法线(o.Normal)参数，将包含这个参数。为了获得基于每个顶点法线贴图( per-pixel normal map)的法线向量(normal vector)需要使用世界法线向量(WorldNormalVector (IN, o.Normal))。

其中的Input结构体，如***上面***三个函数中的Input结构体

**常用结构内容如下：**

struct

{

**形式必须为 uv（小写）加上 面板属性名 （根据命名写全名，有”\_“不要忘写“\_”）**

uv\_MainTex,

**//也可设置为uv2表示次级纹理，其中\_MainTex为属性（properties中定义的名称））**

uv\_BumpTex

uv\_Detail

...

**float3 viewDir,//视角方向，常用于计算边缘光照**

**...还有一些Input的内置结构变量**

**float4 color : COLOR （语义在顶点&片元着色器中的结构体定义中 使用较多）**

};

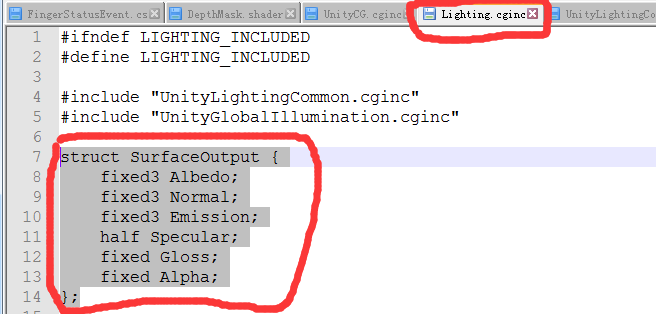
##### 输出：SurfaceOutput SurfaceOutputStandard SurfaceOutputStandardSpecular

其中SurfaceOutput SurfaceOutputStandard SurfaceOutputStandardSpecular 为Unity内置的结构体（inout表示可作为输入输出），可以在光照模型扩展文件中找到：

***SurfaceOutput***

1. **struct** SurfaceOutput
2. {
3. half3 Albedo; // 纹理颜色值（r, g, b)，反射率
4. half3 Normal; // 法向量(x, y, z)  ，法线
5. half3 Emission; // 自发光颜色值(r, g, b)
6. half Specular; // 镜面反射度
7. half Gloss; // 光泽度
8. half Alpha; // Alpha不透明度
9. };

**查找路径如下：**

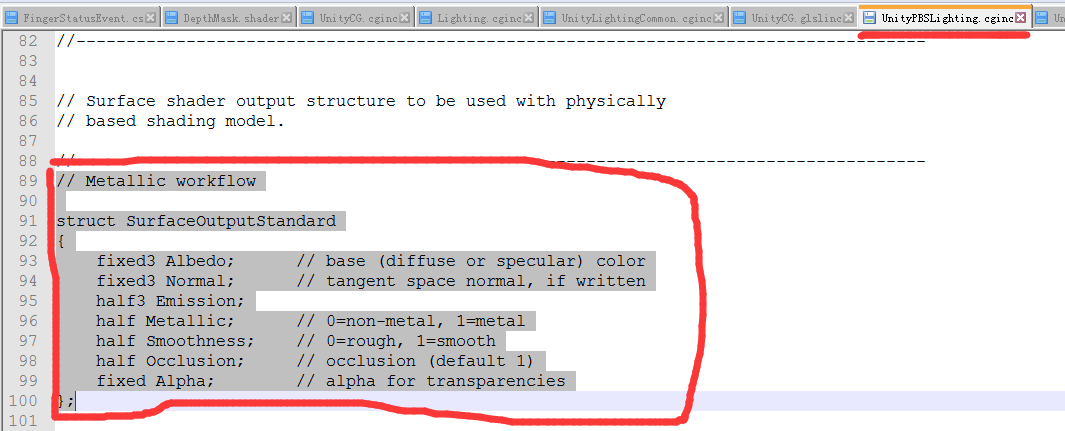


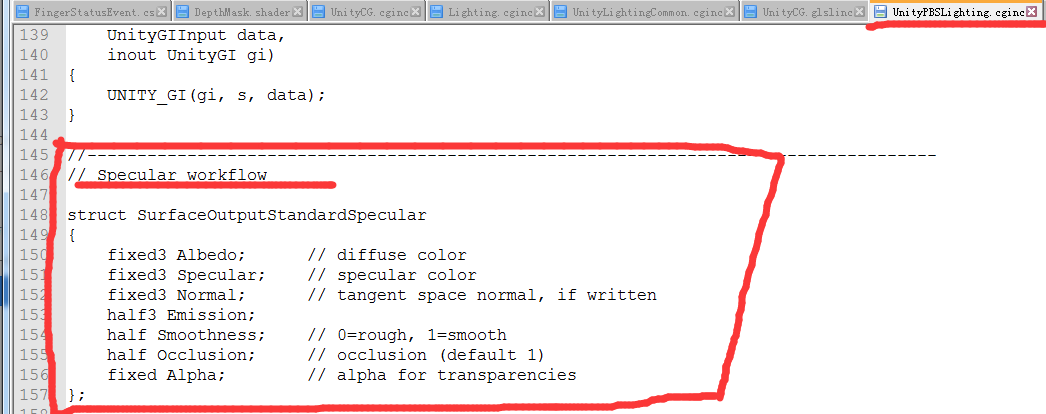
**像反射率的赋值：**

1. //表面着色函数的编写
2. **void** surf (Input IN, inout SurfaceOutput o)
3. {
4. //反射率，也就是纹理颜色值赋为(0.6, 0.6, 0.6)
5. o.Albedo= 0.6;
6. }

***SurfaceOutputStandard | | SurfaceOutputStandardSpecular***

**查找路径：Unity 5.x新添加的**





**【3】**由【2】中的几种SurfaceOutput结构体可知，Unity存在多种光照模型：

1、Unity5.x之前的简单的、不基于物理的光照模型，包括Lambert（兰伯特光照模型，Diffuse）和BlinnPhong（Specular）

2、Unity5.x之后的基于物理的光照模型，包括Standard（默认的金属工作流程，对应光照函数为Standard）

和StandardSpecular（基于高光工作流程，对应的光照函数为StandardSpecular）

3、当然，也可以自定义光照函数，如MyCalc，则在新建后缀为cginc或者在unity安装目录下的CGInclude文件夹中原有的文件中添加新定义的光照函数

声明类似于：

half4 **Lighting**MyCalc(SurfaceOutputs,参数省略){ } **注意：需要在自定义的函数名前加上“Lighting”**

**【4】其他可选参数，待补充了解**

* alpha -透明( Alpha)混合模式。使用它可以写出半透明的着色器。
* alphatest:VariableName -透明( Alpha)测试模式。使用它可以写出 镂空效果的着色器。镂空大小的变量(VariableName)是一个float型的变量。
* vertex:VertexFunction - 自定义的顶点函数(vertex function)。相关写法可参考Unity内建的Shader：树皮着色器(Tree Bark shader)，如Tree Creator Bark、Tree Soft Occlusion Bark这两个Shader。
* **finalcolor:ColorFunction - 自定义的最终颜色函数(final color function)**

**注意： 此处区分大小写 finalColor X**

**常见书写格式：**

**#pragma surface surf Lambert finalcolor:setcolor**

**void setcolor(Input IN, SurfaceOutput o, inout fixed4 color)**

**//注意：\*\*\*需要三个参数\*\*\***

**{**

**color \*= \_ColorTint;**

**}**

* **其他：**
* exclude\_path:prepass 或者 exclude\_path:forward - 使用指定的渲染路径，不需要生成通道。
* addshadow - 添加阴影投射 & 收集通道(collector passes)。通常用自定义顶点修改，使阴影也能投射在任何程序的顶点动画上。
* dualforward - 在[正向(forward)](http://game.ceeger.com/Components/RenderTech-ForwardRendering.html)渲染路径中使用 [双重光照贴图(dual lightmaps)](http://unity3d.com/support/documentation/Manual/LightmappingInDepth.html#DualLightmaps)。
* fullforwardshadows - 在[正向(forward)](http://game.ceeger.com/Components/RenderTech-ForwardRendering.html)渲染路径中支持所有阴影类型。
* decal:add - 添加贴图着色器(decal shader) (例如： terrain AddPass)。
* decal:blend - 混合半透明的贴图着色器(Semitransparent decal shader)。
* softvegetation - 使表面着色器(surface shader)仅能在Soft Vegetation打开时渲染。
* noambient - 不适用于任何环境光照(ambient lighting)或者球面调和光照(spherical harmonics lights)。
* novertexlights - 在正向渲染(Forward rendering)中不适用于球面调和光照(spherical harmonics lights)或者每个顶点光照(per-vertex lights)。
* nolightmap - 在这个着色器上禁用光照贴图(lightmap)。(适合写一些小着色器)
* nodirlightmap - 在这个着色器上禁用方向光照贴图(directional lightmaps)。 (适合写一些小着色器)。
* noforwardadd - 禁用正向渲染添加通道(Forward rendering additive pass)。 这会使这个着色器支持一个完整的方向光和所有光照的per-vertex/SH计算。(也是适合写一些小着色器).
* approxview - 着色器需要计算标准视图的每个顶点(per-vertex)方向而不是每个像索(per-pixel)方向。 这样更快，但是视图方向不完全是当前摄像机(camera) 所接近的表面。
* halfasview - 在光照函数(lighting function)中传递进来的是half-direction向量，而不是视图方向(view-direction)向量。 Half-direction会计算且会把每个顶点(per vertex)标准化。这样做会提高执行效率，但是准确率会打折扣。
* *此外，还可以在 CGPROGRAM块内编写 #pragma debug，然后表面编译器(surface compiler)会进行解释生成代码。*

**结合上面四点，编译指令可以如下形式：**

***#pragma surface surf Lambert finalcolor:mycolor。***

*应用：待补充…*

### 顶点着色器&片元着色器

***Vertex Shader & Fragment Shader：***

与DirectX的HLSL和CG紧密相连，Unity中就是用ShaderLab进行了包装。

就渲染管线中的概念总结：

顶点着色器：产生纹理坐标，颜色，点大小，雾坐标，然后把它们传递给裁剪阶段。

片段着色器：进行纹理查找，决定什么时候执行纹理查找，是否进行纹理查找，及把什么作为纹理坐标。

**相较于表面着色器，该着色器具有更大的可控性和可编程性，可实现更丰富效果，最接近CG着色语言**

**注意：由于顶点着色器一般是渲染管线的最开始，所以Unity预定义了一些常用的输入数据结构**

**（下面列出的是针对顶点着色器的输入结构：注意区分表面着色器的输入结构input）**

|  |  |
| --- | --- |
| **数据结构** | **含义** |
| appdata\_base | 顶点着色器输入位置、法线以及一个纹理坐标。 |
| appdata\_tan | 顶点着色器输入位置、法线、切线以及一个纹理坐标。 |
| appdata\_full | 顶点着色器输入位置、法线、切线、顶点颜色以及四组纹理坐标。（最大6个纹理坐标） |
| appdata\_img | 顶点着色器输入位置以及一个纹理坐标。 |

**具体查找路径：...\Editor\Data\CGIncludes\unityCG.cginc**

#### Unity支持的语义：

从应用阶段传递模型数据到顶点着色器时支持的语义

|  |  |
| --- | --- |
| 语义 | 描述 |
| POSITION | 模型空间中的顶点位置，通常float4类型 |
| NORMAL | 顶点法线，通常为float3类型，即（x,y,z）三个元素组成的矢量 |
| TANGENT | 顶点切线，通常为float4类型,(x,y,z,w) |
| TEXCOORDn,如TEXCOORD0、Texcoord1 | 顶点的纹理坐标，0表示第一组纹理坐标 |
| COLOR | 顶点颜色，通常为fixed4或float4类型 |

从顶点着色器传递数据给片元着色器时支持的语义

|  |  |
| --- | --- |
| 语义 | 描述 |
| SV\_POSITION | 裁剪空间的顶点坐标，结构体中必须包含一个用该语义修饰的变量，在DirectX 9 中使用POSITION,推荐使用SV\_POSITION |
| COLOR0 | 通常用于输出第一组的顶点颜色，非必须，**以下三项不一定仅存储当前描述的变量** |
| COLOR1 | 通常用于输出第二组顶点颜色，非必须 |
| TEXCOORD0~TEXCOORD7 | 通常用于输出纹理坐标，非必须 |

片元着色器输出时支持语义

|  |  |
| --- | --- |
| 语义 | 描述 |
| SV\_Target | 输出值将会存储到渲染目标中，等同于Direct9 中的COLOR,推荐SV\_Target |

说明：

像fixed4 float4这类存储了4个浮点值的变量，如果需要存储4X4的矩阵，可以定义4组float4来满足需求

**如果想让顶点着色器和片元着色器进行通信，需要保证顶点着色器传递了一个语义为（POSITION或者SV\_POSITION）的变量，其中包含了顶点坐标**

语义：

特殊情况

SV\_POSITION 替换 POSITION（由于不支持PS4）

SV\_Target 替换 COLOR（由于不支持PS4）

SV的含义：系统数值（system-value）

**//使用COLOR语义，作为从Vertex阶段到Fragment阶段的输入输出的约定**

**其他语义还有**

**SV\_POSITION**

**POSITION**

关于上面两者的区别：<http://blog.csdn.net/zhao_92221/article/details/46797969>

<http://forum.unity3d.com/threads/what-is-the-difference-between-float4-pos-sv_position-and-float4-pos.165351/>

**NORMAL**

**TEXCOORD[n] : 如TEXCOORD1**

**具体可见HLSL语义库（不仅仅只是Unity支持的语义）：**

[**https://msdn.microsoft.com/en-us/library/windows/desktop/bb509647.aspx**](https://msdn.microsoft.com/en-us/library/windows/desktop/bb509647.aspx)

#### Vertex shader & fragment shader 大致形式：

Shader "Inspect面板显示名称/子菜单名称"{

Properties{

\_Color("Inspect中的Shader面板显示名称"，Color) = (1,1,1,1)

........

}

SubShader{

pass{

CGPROGRAM

//定义顶点着色器和片元着色器的函数名

#pragma vertex vert

#pragma fragment frag

//引用后缀为 .cginc 的文件 包含了一些常用的函数和宏

#include "unitycg.cginc"

//声明结构体 （v2f：用于顶点着色器传递信息给片元着色器）

struct v2f{

float4 pos : POSITIVE;//或者使用SV\_POSITION，pos里包含了顶点在裁剪空间的坐标

fixed4 color : COLOR; //存储颜色信息

//Unity支持的语义有POSITION,TANGENT(切线),NORMAL(法线)

//Texcoord0,1,2,3等，COLOR等

//（SV\_为HLSL的系统语义，常用于输出）,SV\_POSITION获取

//可用于光栅化的变换后的顶点坐标（即齐次裁剪空间中的坐标）

};

//声明面板属性（uniform）

(uniform)float4 \_Color;

v2f vert(appdata\_base v) //顶点着色器

{

v2f o;

o.pos = mul(UNITY\_MATRIX\_MVP,v.vertex);//将输入的模型顶点从模型空间变换到裁剪空间

o.color = fixed4(1,1,1,1); //float4 fixed4 half4仅仅表示精度，精度有高到低

return o; //输出顶点颜色

}

fixed4 frag(v2f IN) : COLOR //片元着色器中的输入实际上是把顶点着色器的输出进行插值的结果

{

...... //片元着色器：

return IN.color; //输出像素颜色

}

ENDCG}

Fallback”回调”}}

*应用：待补充…*

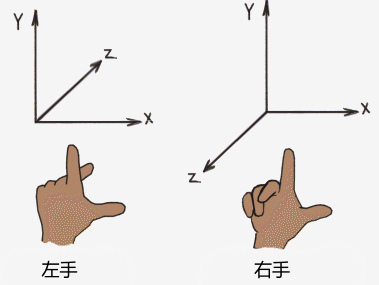
## Unity 使用的坐标系

### 渲染管线中模型顶点的空间变换过程（一般在顶点着色器中实现）

顶点着色器帮助我们将模型顶点从模型空间一步步转换到裁剪空间，在Unity中，使用UNITY\_MATRIX\_MVP这个变换矩阵方法可以快速实现转换，裁剪空间到屏幕空间由Unity内部实现。



### 左&右手坐标系的定义



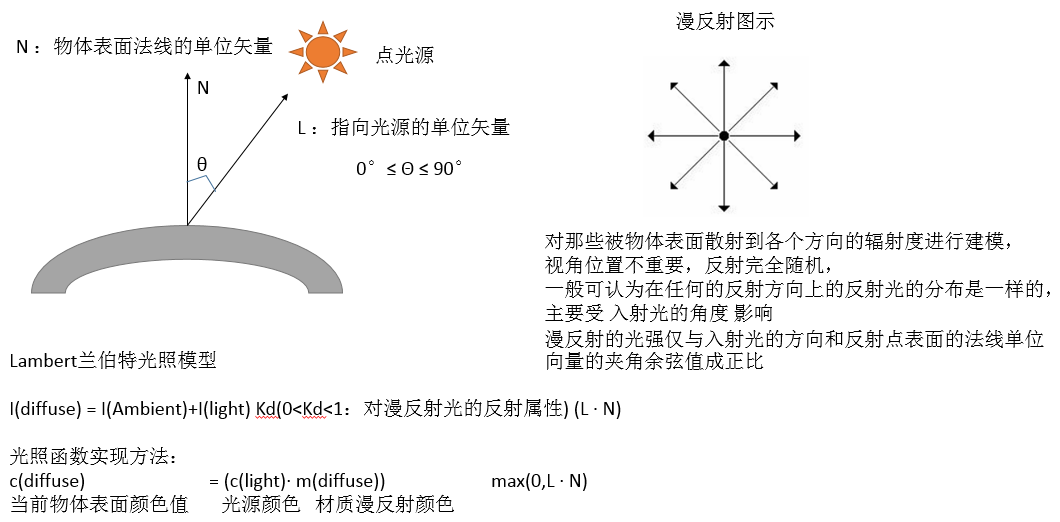
## Unity中的基础光照

**标准光照模型**

根据材质属性（漫反射、镜面反射等）、光照信息（光源方向、辐射度等），使用一个等式去计算沿某个观察方向的出射度的过程，这个过程可称之为光照模（Lighting Model）

**· 漫反射**

**Lambert**



unity内置Lighting.cginc中给出的Lambert光照模型 部分参考代码：

fixed4 \_LightColor0;

**inline** fixed4 LightingLambert (SurfaceOutput s, fixed3 lightDir, fixed atten)

{

    fixed diff = max (0, dot (s.Normal, lightDir));

    fixed4 c;

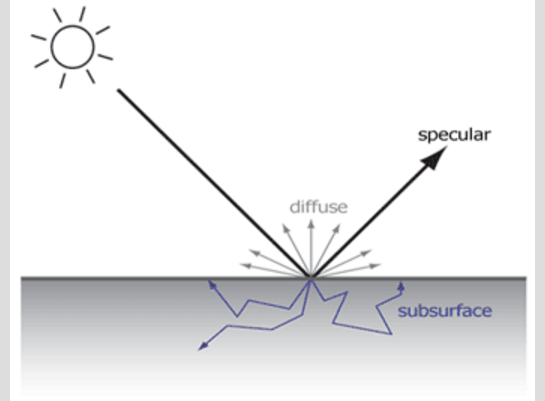
    c.rgb = s.Albedo \* \_LightColor0.rgb \* (diff \* atten \* 2);

    c.a = s.Alpha;

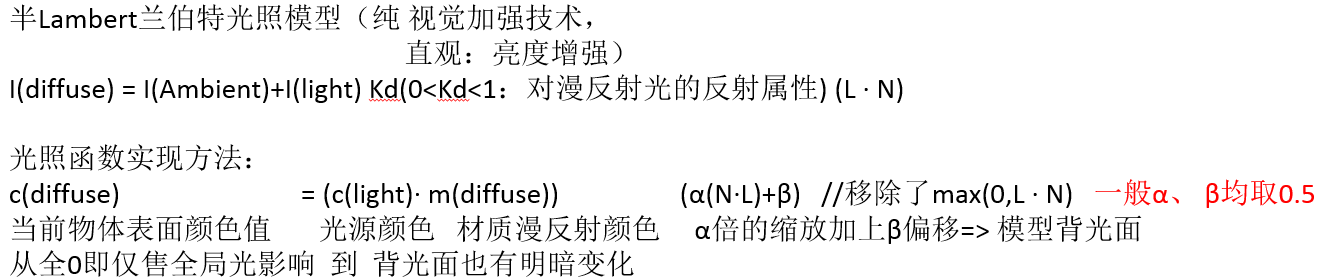
**return** c;

}

**扩展：半兰伯特光照模型（视觉加强技术）：为Subsuface Scattering(表面散射)**

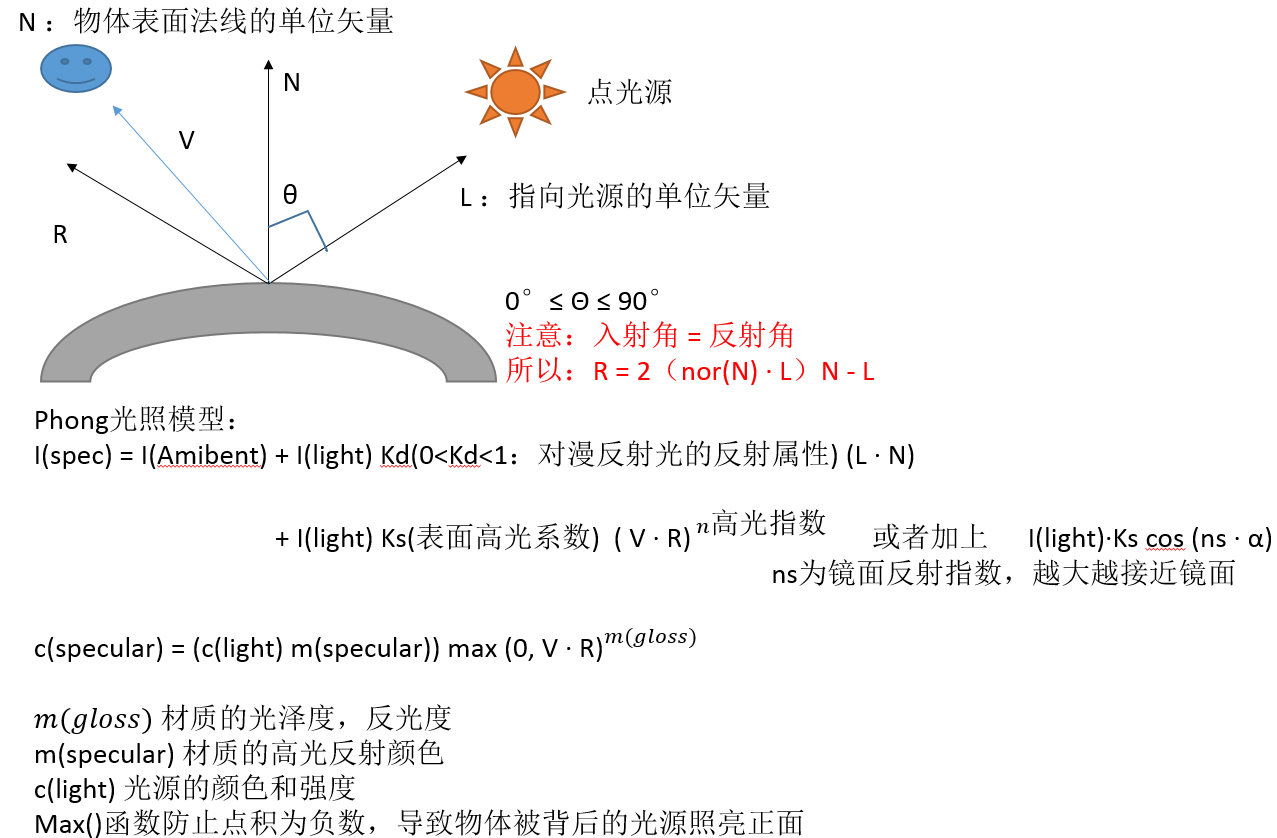




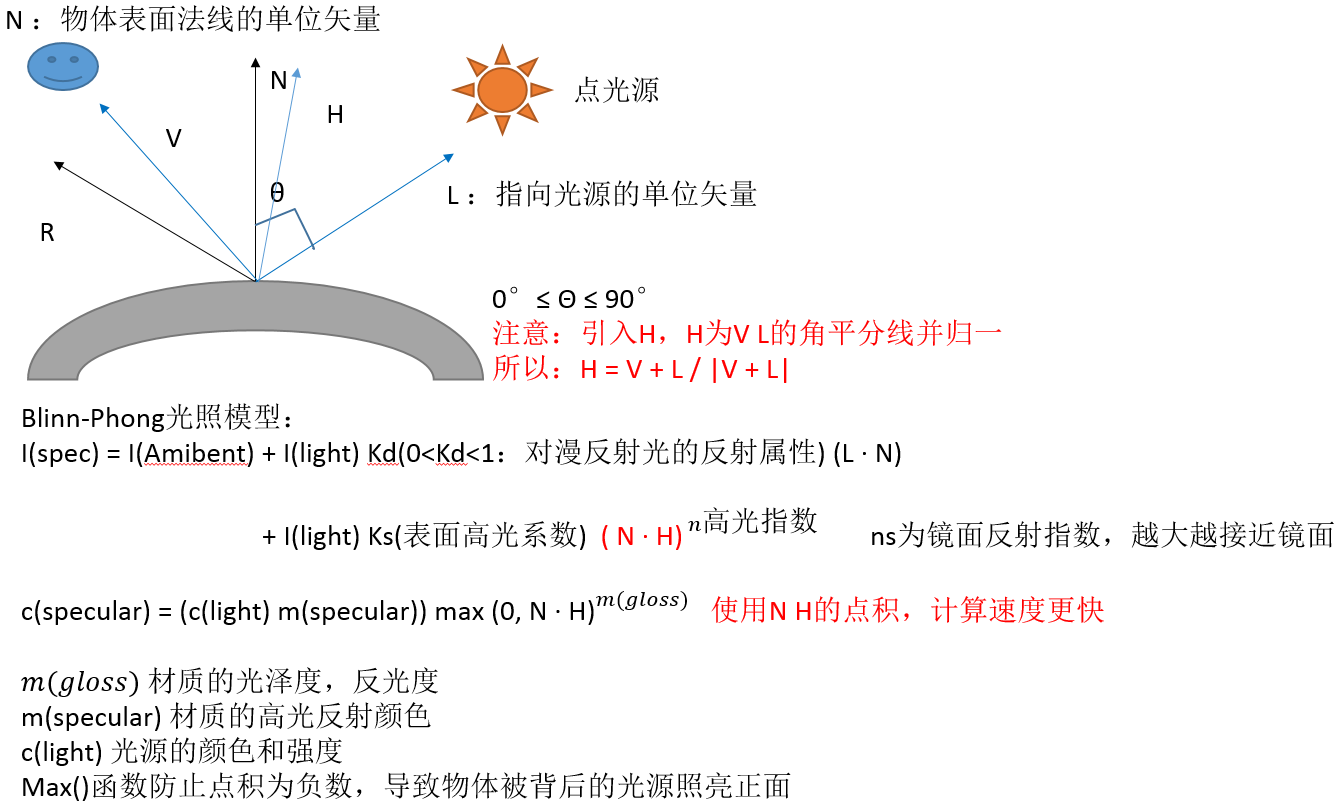


**· 镜面反射**

**Phong光照模型**



**扩展：Blinn-Phong光照模型**



具体实现见 CGInclude中的Lighting.cginc

大致浏览表面着色器和顶点&片元着色器分别实现一个光照模型的shader结构

**表面着色器光照模型及自定义光照模型**

半兰伯特：

1. Shader "Half\_Lambert\_Lighting"
2. {
3. //--------------------------------【属性】----------------------------------------
4. Properties
5. {
6. \_MainTex ("【主纹理】Texture", 2D) = "white" {}
7. }
9. //--------------------------------【子着色器】----------------------------------
10. SubShader
11. {
12. //-----------子着色器标签----------
13. Tags { "RenderType" = "Opaque" }
14. //-------------------开始CG着色器编程语言段-----------------
15. CGPROGRAM
17. //【1】光照模式声明：使用自制的半兰伯特光照模式
18. #pragma surface surf QianMoHalfLambert
20. //【2】实现自定义的半兰伯特光照模式
21. half4 LightingQianMoHalfLambert (SurfaceOutput s, half3 lightDir, half atten)
22. {
23. half NdotL =max(0, dot (s.Normal, lightDir));
25. //在兰伯特光照的基础上加上这句，增加光强
26. **float** hLambert = NdotL \* 0.5 + 0.5;
27. half4 color;
29. //修改这句中的相关参数
30. color.rgb = s.Albedo \* \_LightColor0.rgb \* (hLambert \* atten \* 2);
31. color.a = s.Alpha;
32. **return** color;
33. }
35. //【3】输入结构
36. **struct** Input
37. {
38. float2 uv\_MainTex;
39. };
41. //变量声明
42. sampler2D \_MainTex;
44. //【4】表面着色函数的编写
45. **void** surf (Input IN, inout SurfaceOutput o)
46. {
47. //从主纹理获取rgb颜色值
48. o.Albedo = tex2D (\_MainTex, IN.uv\_MainTex).rgb;
49. }
51. //-------------------结束CG着色器编程语言段------------------
52. ENDCG
53. }
55. Fallback "Diffuse"
56. }

**顶点&片元着色器实现光照模型**

顶点着色器：逐顶点光照（计算量较少）

片元着色器：逐像素光照

单色镜面反射：（Phong光照），逐像素的光照

1. Shader "浅墨Shader编程/Volume13/4.Specular"
2. {
3. //------------------------------------【属性值】------------------------------------
4. Properties
5. {
6. //主颜色
7. \_Color("Main Color", Color) = (1, 1, 1, 1)
8. //镜面反射颜色
9. \_SpecColor("Specular Color", Color) = (1, 1, 1, 1)
10. //镜面反射光泽度
11. \_SpecShininess("Specular Shininess", Range(1.0, 100.0)) = 10.0
12. }
14. //------------------------------------【唯一的子着色器】------------------------------------
15. SubShader
16. {
17. //渲染类型设置：不透明
18. Tags{ "RenderType" = "Opaque" }

21. //--------------------------------唯一的通道-------------------------------
22. Pass
23. {
24. //光照模型ForwardBase
25. Tags{ "LightMode" = "ForwardBase" }
26. //===========开启CG着色器语言编写模块===========
27. CGPROGRAM
29. //编译指令:告知编译器顶点和片段着色函数的名称
30. #pragma vertex vert
31. #pragma fragment frag
33. //顶点着色器输入结构
34. **struct** appdata
35. {
36. float4 vertex : POSITION;//顶点位置
37. float3 normal : NORMAL;//法线向量坐标
38. };
40. //顶点着色器输出结构
41. **struct** v2f
42. {
43. float4 pos : SV\_POSITION;//像素位置
44. float3 normal : NORMAL;//法线向量坐标
45. float4 posWorld : TEXCOORD0;//在世界空间中的坐标位置
46. };

49. //变量的声明
50. float4 \_LightColor0;
51. float4 \_Color;
52. float4 \_SpecColor;
53. **float** \_SpecShininess;
55. //--------------------------------【顶点着色函数】-----------------------------
56. // 输入：顶点输入结构体
57. // 输出：顶点输出结构体
58. //---------------------------------------------------------------------------------
59. //顶点着色函数
60. v2f vert(appdata IN)
61. {
62. //【1】声明一个输出结构对象
63. v2f OUT;
65. //【2】填充此输出结构
66. //输出的顶点位置为模型视图投影矩阵乘以顶点位置，也就是将三维空间中的坐标投影到了二维窗口
67. OUT.pos = mul(UNITY\_MATRIX\_MVP, IN.vertex);
68. //获得顶点在世界空间中的位置坐标
69. OUT.posWorld = mul(\_Object2World, IN.vertex);
70. //获取顶点在世界空间中的法线向量坐标
71. OUT.normal = mul(float4(IN.normal, 0.0), \_World2Object).xyz;
73. //【3】返回此输出结构对象
74. **return** OUT;
75. }
77. //--------------------------------【片段着色函数】-----------------------------
78. // 输入：顶点输出结构体
79. // 输出：float4型的像素颜色值
80. //---------------------------------------------------------------------------------
81. fixed4 frag(v2f IN) : COLOR
82. {
83. //【1】先准备好需要的参数
84. //获取法线的方向
85. float3 normalDirection = normalize(IN.normal);
86. //获取入射光线的方向
87. float3 lightDirection = normalize(\_WorldSpaceLightPos0.xyz);
88. //获取视角方向
89. float3 viewDirection = normalize(\_WorldSpaceCameraPos - IN.posWorld.xyz);
91. //【2】计算出漫反射颜色值  Diffuse=LightColor \* MainColor \* max(0,dot(N,L))
92. float3 diffuse = \_LightColor0.rgb \* \_Color.rgb \* max(0.0, dot(normalDirection, lightDirection));
94. //【3】计算镜面反射颜色值
95. float3 specular;
96. //若是法线方向和入射光方向大于180度，镜面反射值为0
97. **if** (dot(normalDirection, lightDirection) < 0.0)
98. {
99. specular = float3(0.0, 0.0, 0.0);
100. }
101. //否则，根据公式进行计算 Specular =LightColor \* SpecColor \*pow(max(0,dot(R,V)),Shiness),R=reflect(-L,N)  **//光源方向取反**
102. **else**
103. {
104. float3 reflectDirection = reflect(-    lightDirection, normalDirection);
105. specular = \_LightColor0.rgb \* \_SpecColor.rgb \* pow(max(0.0, dot(reflectDirection, viewDirection)), \_SpecShininess);
106. }
108. //【4】合并漫反射、镜面反射、环境光的颜色值
109. float4 diffuseSpecularAmbient = float4(diffuse, 1.0) + float4(specular, 1.0) + UNITY\_LIGHTMODEL\_AMBIENT;
111. //【5】将漫反射-镜面反射-环境光的颜色值返回
112. **return** diffuseSpecularAmbient;
113. }
115. //===========结束CG着色器语言编写模块===========
116. ENDCG
117. }
118. }
119. }

## 基础纹理（待补充）

### 纹理uv

### 凹凸映射

### 渐变纹理

### 遮罩纹理

### 顶点动画

待补充

## 高级光照（待补充）

### 阴影，透明度物体的阴影正确显示

### Unity的渲染路径（Rendering Path）

#### 前向渲染路径(Forward)

待补充

#### 延迟渲染路径(Deferred)

待补充

#### 顶点照明渲染路径(Vertex Lit)

待补充

## 高级纹理（待补充）

### 噪声纹理

### 深度纹理、法线纹理

#### 运动模糊

#### 全局雾效

#### 边缘检测

##### 卡通风格

##### 素描风格

### 程序纹理

#### 程序材质（Procedural Materials）

使用Substance Designer软件进行制作，可以在Asset store中搜索Substance材质（后缀为.sbsar）

## 屏幕后期处理（待补充）

Image Effect（图像特效）(c#脚本&shader)

主要应用在Camera摄像机对象上。Unity中所有的图像特效抖编写再OnRenderImage函数中，任何附加在摄像机对象上的Image Effect脚本都可以通过编辑其代码来修改特效。

使用参数详解：

1、Antialiasing （Fullscreen）：抗锯齿（全屏）特效

      此特效提供了平滑图像的功能。图形硬件渲染出的多边形的边缘通常会有锯齿，影响视觉效果，而全屏抗锯齿特效能够平滑这些锯齿，增强场景的视觉效果。通常情况下，抗锯齿效果的质量与算法的速度成反比，该特效的属性面板如图：

Technique：抗锯齿技术。该项用于选择抗锯齿的方式，

－FXAA2:快速近似抗锯齿算法。

－FXAA3 Console：快速近似抗锯齿算法控制台，该项为默认选项。

       Edge Min Threshold：边缘最小阀值。

       Edge Threshold：  边缘阀值。

       Edge Sharpness：  边缘清晰度。

－FXAA1 PresetA：预设快速近似抗锯齿算法A。

－FXAA1 PresetB：预设快速近似抗锯齿算法B。

－NFAA：仅模糊局部边界的边缘模糊算法。

  Edge Detect Ofs：边缘检测。

  Blur Radius：模糊半径。

  Show Normals： 选中此项则显示为法线效果。

－SSAA：仅模糊局部边界的边缘模糊算法。

－DLAA：自适应处理长边界抗锯齿算法。

   Sharp：清晰度。

2、泛光特效

     泛光可以理解为是一种增强版的辉光、眩光效果。泛光效果能在增加光晕的同时自动添加高效能镜头眩光。Bloom是一个非常独特的效果，在HDR渲染（高动态光照渲染）的情况下可能为场景添加梦幻般的感觉。适当的调整能使画面增强真实感，比如当光照对比差异悬殊的情况下，明亮的部分会显得像在发光，类似摄影时的效果，该特效属性如下：

－Quality：质量。该项用于选择质量等级，有两种等级可供选择。

   Cheap：低等级质量。计算速度比较快。

   High：高等级质量。该项为默认选项。

－Mode：模式。该项用于选择模式，有两种模式可供选择。

    Basic：基本模式。

    Complex：复杂模式。

－Blend：混合。该项用于选择混合方式，有两种方式可供选择。

    Screen：屏幕模式。该模式模拟两种图像同时投射到屏幕上。每个颜色通道被分开处理和渲染。相比Add（叠加模式），能保留更多颜色变化和细节。

    Add：叠加模式。该模式下，R、G、B通道的各个颜色值累加，最大值为1，可以使亮度较低的像素变亮，当需要实现耀眼的白色光晕时使用该模式。

－HDR：高动态光照渲染。该项用于控制HDR的开关，有3种方式可供选择。

  Auto：自动，该项为默认选项。该项用于根据摄像头的HDR设置来控制高动态光照渲染效果的开关。

  On：强制打开高动态光照渲染效果。

  Off：强制关闭高动态光照渲染效果。

－Intensity：强度。用于控制全局光照的强度，主要影响泛光和光晕。

－Threshhold：阀值。用于控制泛光和光晕计算的阀值。

－RGB Threshhold：RGB阀值。只在Mode设置为Complex模式下生效，可以为每个颜色通道分别设置阀值。

－Blur Iterations：模糊迭代。即重复应用多少次搞死模糊到图像上。迭代次数越多效果越平滑，但同时会花费更多时间。

－Sample Distance：采样间距。该项用于控制最大模糊半径，对性能影响较小。

－Lens Flares：镜头眩光。该项只在Mode设置为Complex模式下有效，有3种方式可供选择。

  Ghosting：重影镜头眩光类型。

  Anamorphic：变形镜头眩光类型。

  Combined：组合类型。以上两种镜头眩光类型的结合。

－Local Intensity：局部强大。该项只在Mode设置为Complex模式下有效，此属性仅用于镜头眩光，值为0表示镜头眩光无效果，非0的值则会有如下选项。

  1st－4th Color：颜色调整。仅当镜头眩光模式为重影或混合时有效。

  Stretch width：拉伸宽度，用于控制镜头眩光的变形宽度。

  Rotation：旋转，用于控制镜头眩光的变形方位。

  Blur Iterations：模糊迭代，用于控制变形镜头眩光的模糊迭代次数，次数越高光晕越圆滑，但会花费更多时间。

－Saturation：饱和度，用于控制镜头眩光饱和度，如果值为0，光晕的颜色将近似于Tint Color。

－Tint Color：着色，用于调整变形镜头眩光的颜色。

－Mask：光晕遮蔽图，指定一张图片作为遮蔽图，用于实现屏幕边缘的镜头眩光效果。

－Threshold：局部阀值。定义了哪部分图像将被用于产生镜头眩光。

3、Bloom And Flares：泛光和镜头眩光特效

        Bloom And Flares的特效类型与Bloom特效类似，Bloom And Flares特效的

4、Blur：模糊特效

        图像模糊特效可以实时地将渲染出的游戏画面进行模糊处理，该特效属性面板

5、Color Correction Curves： 色彩校正（曲线）特效

        该特效使用曲线调整每一个颜色通道，也可以根据每个像素的深度进行调整。

6、Contrast Enhance：对比度增强特效

        对比度增强特效可以增强游戏画面的对比度，其原理是使用了图像处理领域中非锐化遮蔽图方式来达到增强对比度的效果。

7、Edge Detection： 几何边缘检测特效

        边缘检测的图像特效是根据场景中游戏对象的几何形状来绘制其轮廓线。边缘由颜色的差异、相邻像素所对应的法线朝向以及深度等因素共同来决定。

8、 Depth Of Field Deprecated：  景深特效

        景深特效是常见的模拟摄像机聚焦效果的图像特效。现实生活中，摄像机只可以聚焦特定距离的物体；更近或更远的物体将会出现一定的散焦现象。景深特效不仅提供了一个关于物体距离的视觉提示，同时也带来背景虚化效果。

## Unity渲染优化

Unity的优化方案：大致可分为CPU端和GPU端的优化

1. CPU

过多的DrawCall （解决方案：使用批处理）

复杂的脚本和物理模拟

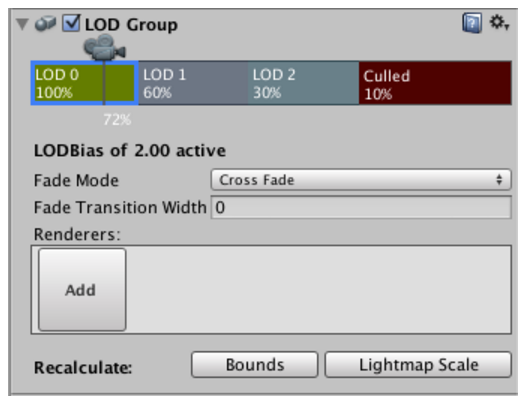
1. GPU

· 顶点处理

过多的顶点 、过多的逐顶点计算

（解决方案：优化几何体的顶点数、使用LOD技术、使用遮挡剔除（Occlusion Culling））

LOD：（纹理Mesh层）：通过添加LOD Group组件，对指向同一物体对象的不同大小的Mesh进行分层



·片元处理

过多的片元（分辨率或者overDraw（一个像素被绘制多次））

过多的逐片元计算

（解决方案：控制绘制顺序、小心处理透明物体、减少实时光照；减少计算复杂度）

减少计算复杂度：

使用shader的LOD（Level of Detail）设置最小可被使用的shader层；代码优化

1. 带宽

使用了尺寸过大且未压缩的纹理（减少纹理大小）

分辨率过高的帧缓存（利用分辨率缩放）

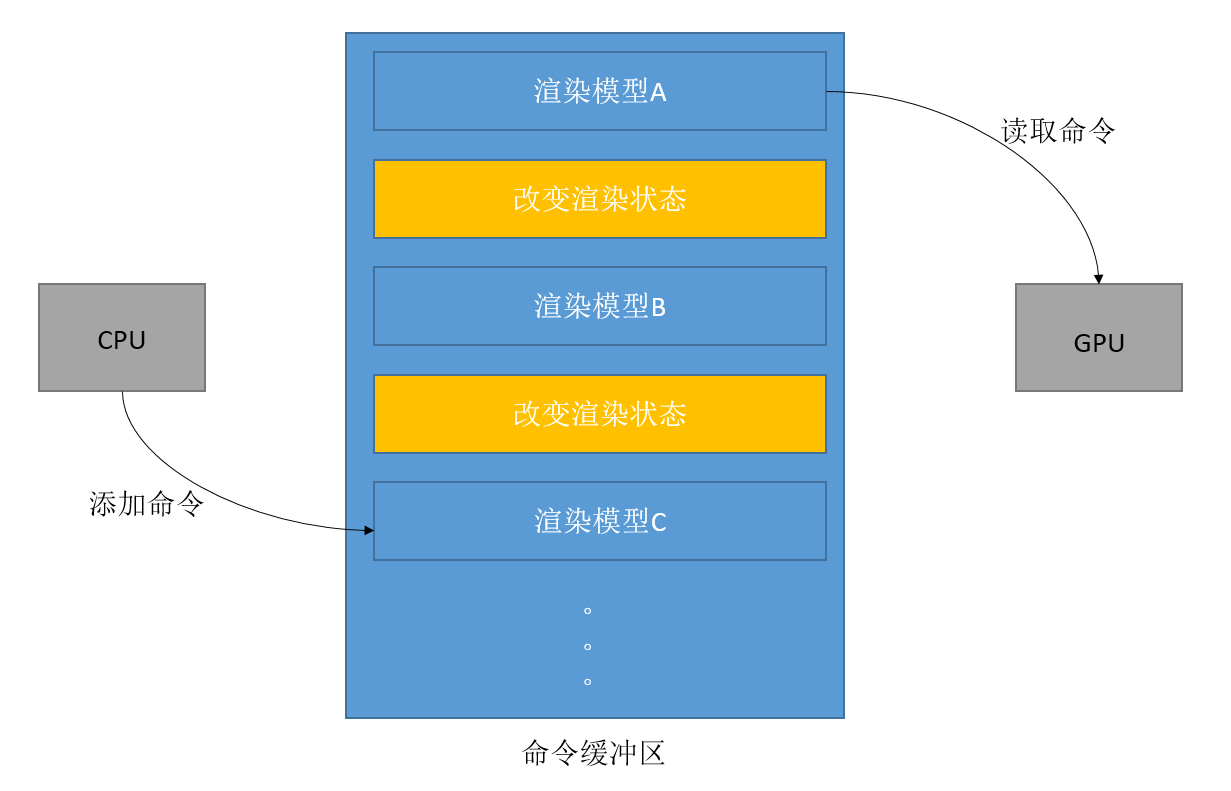
### DrawCall

#### 什么是DrawCall

DrawCall就是CPU调用图像编程接口，如OpenGL中的glDrawElements命令或者DirectX中的DrawIndexedPrimitive命令，以命令GPU进行渲染操作

先了解一下渲染指令的发送：

**· CPU和GPU实现并行工作**



上图中黄色框的命令比较耗时

**· Draw Call 多了会影响效率**

一般情况下，GPU的渲染速度远远快于CPU提交命令的速度，GPU对于渲染200个还是2000个三角网格通常没有什么区别，然而CPU发送命令包含了很多内容，需要完成如检查渲染状态等的很多准备工作，Draw Call数量很多时，CPU就会把大量时间花费在提交Draw Call上，造成**CPU过载**。

#### 如何减少DrawCall

批处理（动态和静态）

注意：避免使用大量很小的网格

避免使用过多的材质

动态批处理的限制条件：基于Unity 5.x以上

1、能够进行动态批处理的网格的顶点属性规模要小于900

2、4.x版本中要求动态批处理的对象需要使用同一个缩放尺度

3、使用光照纹理（lightmap）的物体需要保证均指向光照纹理中的同一位置

4、由于为模型添加更多光照效果需要使用额外的pass通道，而多pass的shader会中断批处理；注意只有物体在点光源的影响范围内才需要调用额外的pass来处理

静态批处理：需要占用更多的内存来存储合并后的几何结构

实现方式：勾选上Inspector面板上的Static复选框中的 **Batching Static**

## 基于物理的渲染

### Unity5中的Standard Shader

Standard shader提供了一个万能的shader，调整少量参数渲染大多数材质；包括了Standard（金属工作流）和Standard(Specular)工作流；可在内建的CGIncludes文件中找到类似于“UnityStanardXXX.cginc”的实现文件

PBS需要配合较复杂的光照来配合，如Light Probe和Reflection Probe，也需要HDR（高动态范围：亮度物体可以非常亮，暗的物体非常暗，且两者之间的细节明显，但渲染速度很慢），也需要添加屏幕特效（抗锯齿、泛光等）；

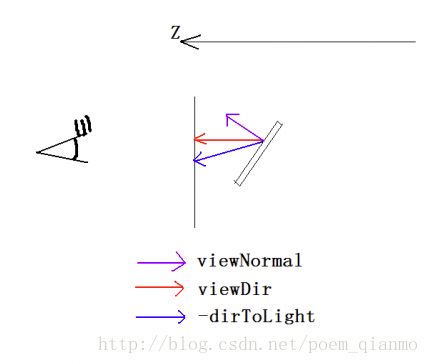
PBS需要创建更细腻复杂的纹理集（金属纹理，高光反射纹理，粗糙度纹理，遮罩纹理等）

## 更多实现（待深入）

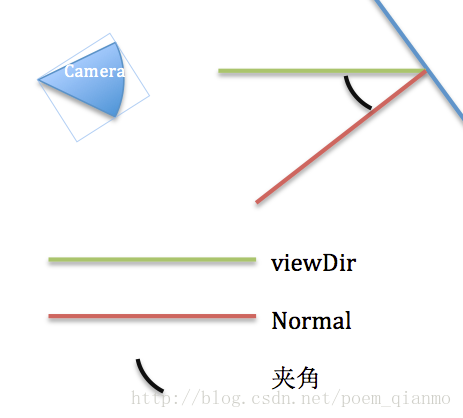
### 边缘光实现

如下图所示：

其中的viewDir 意为WorldSpace View Direction，也就是当前坐标的视角方向：



使用Normalize函数，用于获取到的viewDir坐标转成一个单位向量且方向不变，外面再与点的法线做点积。最外层再用 saturate算出一个[0,1]之间的最靠近的值。这样算出一个rim边界。原理可以这样解释：



这里o.Normal就是单位向量。外加Normalize了viewDir。因此求得的点积就是夹角的cos值。因为cos值越大，夹角越小，所以，这时取反操作。这样，夹角越大，所反射上的颜色就越多。于是就得到的两边发光的效果 ：

#### 表面着色器实现

Shader "学习Shader/Surface/Surf\_RimLighting" {

Properties {

\_MainTex ("Albedo (RGB)", 2D) = "white" {}

\_BumpMap("BumpMap Normal",2D) = "bump"{} //使用了凹凸纹理

\_Detail("Texture Detail", 2D) = "gray"{} //使用了灰度图

//用于自定义表面色泽时使用

\_ColorTint("ColorTint",Color) = (0.6,0.3,0.6,0.3)

//边缘光照 颜色

\_RimColor("Rim Color", Color) = (0.26,0.19,0.16,0.0)

//边缘光照 宽度

\_RimPower("Rim Power",Range(0.5,8.0)) = 3.0

}

SubShader

{

//指明着色器类型， 当渲染非透明物体时调用

Tags { "RenderType"="Opaque" }

LOD 200

CGPROGRAM

//【1】声明光照模式：使用Lambert光照 + 自定义颜色函数

#pragma surface surf Lambert finalcolor:setcolor

//【2】输入结构

struct Input

{

//主纹理的uv坐标

float2 uv\_MainTex;

//凹凸纹理的uv坐标

float2 uv\_BumpMap;

//细节纹理的uv坐标

float2 uv\_Detail; /\* 注意：此处名称需要保证为uv\_面板属性名 否则 最终显示的效果会不一致 \*/

//当前坐标的视角方向

float3 viewDir;

};

//【3】 变量声明

sampler2D \_MainTex;

sampler2D \_BumpMap;

sampler2D \_Detail;

fixed4 \_ColorTint;

float4 \_RimColor;

float \_RimPower;

//【4】自定义颜色函数 用于编译指令的自定义属性

void setcolor(Input IN, SurfaceOutput o, inout fixed4 color)//\*\*\*必须要三个参数

{

color \*= \_ColorTint;

}

//【5】表面着色函数

void surf(Input IN, inout SurfaceOutput o)

{

//从主纹理中获取RGB颜色值

o.Albedo = tex2D(\_MainTex, IN.uv\_MainTex).rgb;

//设置细节纹理

o.Albedo \*= tex2D(\_Detail, IN.uv\_Detail).rgb \* 2;

//从凹凸纹理获取法线值

o.Normal = UnpackNormal(tex2D(\_BumpMap, IN.uv\_BumpMap));

//根据 视线 和 表面法线 获取 不同区域的表面的 自发光 颜色及强度

half rim = 1.0 - saturate(dot(normalize(IN.viewDir), o.Normal));

// saturate : x = Max(0,Min(1,x)) dot 点积

o.Emission = \_RimColor.rgb \* pow(rim, \_RimPower); //指数

}

ENDCG

}

//【6】回滚 普通漫反射

FallBack "Diffuse"

}

#### 顶点&片元着色器实现