本文欢迎转载，但是请注明出处：<https://github.com/lygyue/Books>

纹理

什么是纹理，什么是图片，什么是材质？

纹理和图片的区别在哪里？

随手百度一下，你会搜到一大堆答案。然而这些答案都是些垃圾答案.

我随手放个AOE，估计会被喷死。我不入地狱，谁入地狱！

为什么垃圾答案盛行？很简单，绝大部分人，没入门。中文有一个很明显的特性：那就是望文生义，顾名思义。然而，望文生义，顾名思义，在搞科学的时候，往往会大错特错。科学越是抽象，越是不能望文生义。就好比被吐槽很多的翻译：鲁棒性，上下文，套接字……当你开始望文生义，企图去理解这个东西的时候，你就已经输了。

那么纹理跟图片有什么区别？

我来装个逼：

凡是过往，皆为序章！

凡是图片，皆为纹理！

看到没，一模一样的句式，一模一样的字数，我写的跟大佬写的，云泥之别：（

开个玩笑，图片皆为纹理，估计大家就明白了，纹理反过来就不一定了。

我从程序的角度解释一下：

在图形图像的基础章节里，我有详细讲过：图片，是颜色的量化。那么，储存的必然就是颜色信息。

3D中的纹理，就不一定了，不管你是什么图片格式，储存的不一定是颜色信息。

例如，你可以用JPG来储存高度图，可以储存热力图，储存什么水波纹理图，储存透明通道……

什么？你要打我脸，说我居然不知道JPG不能储存透明通道？

不好意思，我还真不那么容易被打脸。你那么想，只能说明你的思维过于公式化。JPG三个通道，储存的是0-255之间的数据，这些数据，纯图像的时候是RGB，但是在3D里面，可以是任何你希望的东西。可以是距离，可以是法线，可以是Noise，为什么不能是透明值？

理解不了？理解不了，那么你要么安心做一个API程序员，要么转行吧。这都理解不了，图形学你学不深入的，不是这个理解不了，就是那个理解不了。后续的阴影图你还怎么理解？

那么，各位大佬应该理解了，我们常说的图片，储存的就是颜色信息。而纹理，是跟图片一样的格式，但是储存的是各种各样3D里面需要用到的信息。甚至你可以用图片储存矩阵！

因此，抛弃原来那套顾名思义，望文生义的思维方式，直接把纹理理解为一个你从来没听过的术语。在科学研究上，大量的地方你需要这么做。

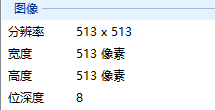
那么，既然这样，那么为什么纹理需要保存跟图片一样的格式呢？都用JPG，PNG，DDS之类的格式来保存呢？这是因为，流水线贯穿于整个GPU的始终。

在认识显卡的章节里，我有提到过，GPU是大量线程的并行计算。而一个常识是：标准化的流水线流程，能最大限度的提高效率。GPU渲染也是如此。所以，图片跟纹理采用同一种格式，我们能做同一种标准化的流水线处理，我们能做一样的压缩算法，我们用一样的储存格式，我们用一样的采样函数，这对于显卡处理效率而言，无疑是最优的。

那么，图片是如何储存例如深度图，高度图，热力图，法线贴图等等一大堆乱七八糟的东西的？其实这部分内容都烂大街了，我不打算深入讲，只讲一个最简单的高度图的储存，其他的以此类推。

假设你要做一个3D游戏里面的地形编辑器，要编辑一个地形，那么先定义地形的大小。根据游戏的规模，大部分是512 \* 512，也有256 \* 256的。当然了，牛逼作品都是无缝地形。其实也是分块的，只是做了拼接。

假设一个512 \* 512的地形，那么顶点数就是513 \* 513。格子数是512，当然要513条边了，这个不难理解。那么根据你游戏的类型，你定一个范围，例如你可以是地形最高200米，最低-200米。那么，你在你的图片里就是-128代码-200，128代表200.随便算算比例即可。例如如果储存的数值是100，那么实际高度就是：(100.0f / 128.0f) \* 200.0f。图片的深度早期都是用的8位。现在估计用32位更合适，可以做到更精细，不差那么一点显存。如图：

这是早期的高度图，用来做地形的。

其他什么热力图，什么深度图……以此类推。

纹理的MipMap。

什么是纹理的mipmap？为什么需要mipmap？

如果没看过我之前的章节：图形图像入门，那么先回去看之前的章节，不然这里看不下去。

之前讲过，图片的压缩没那么简单。放到这里就是：纹理的采样没那么简单。

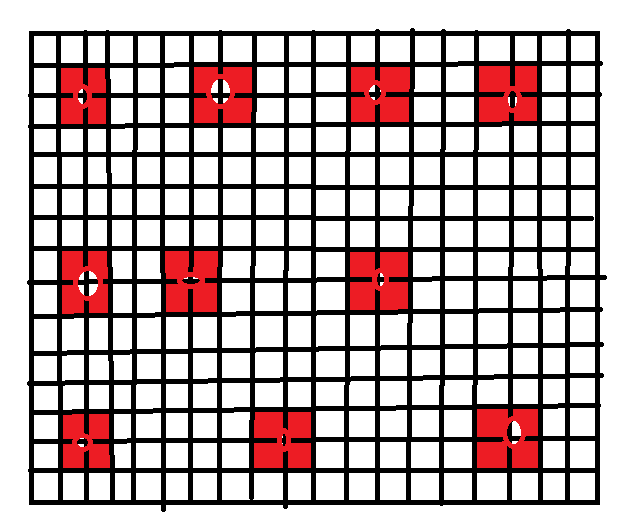
假设一张纹理，分辨率是1024 \* 1024，现在渲染一个点：UV坐标是（0.3， 0.4），那么，我们可以这样算（这里只讨论最常用的线性采样，最近点采样更加简单）：

Float x = u \* texture\_size = 0.3 \* 1024 = 308.2;

Float y = v \* texture\_size = 0.4 \* 1024 = 409.6;

采样计算这里就不复制粘贴了，不会的回去看图形图像基础那个章节。

我们可以轻易得知：采样到的最终结果，只跟（308，409），（309，409），（308，410），（309，410）这四个像素有关。这有问题吗？看起来没有问题。但是，如果你在屏幕上渲染一个矩形，假设在屏幕上的大小是100 \* 100，而你采样了一张1024 \* 1024的贴图，那么显然造成了巨大的颜色的丢失！看图：



假设一个格子是一个像素，红色的圈圈是采样的UV坐标，那么，任意一个像素仅仅采样到周边四个像素的信息。如果一张1024的贴图只采样了100的像素，那么必然周边的颜色信息都丢失了，不信你自己写个程序这样做一个缩放看看。

除了颜色丢失，还有另外一个问题，一般称作mipmap抖动。由于渲染到屏幕上的像素点太小，而采样图片太大，镜头稍稍移动或者是稍稍有什么风吹草动，只要浮点数有一点点的变动，都会导致像素的剧烈抖动。

举例，假设u坐标是0.3，图片分辨率是1024，那么采样到的坐标是：

0.3 \* 1024 = 308.2。

如果0.3变成了0.31呢？那么就变成了：

0.31 \* 1024 = 317.44；

看到了吗？小数点后两位的一个微小变动，采样的像素差了9个，颜色说不定从白色变成了黑色，不闪烁才是怪事。

与之相反的另外一种情形，就是屏幕坐标过大，而采样图片过小。例如画一个quad，屏幕像素是500 \* 500，采样的是100 \* 100的图片，显然会出现很容易理解的马赛克，跟图片放大是一样的。

解决问题的方案：mipmap。

Mipmap的意思是：我保存一大堆图片，根据采样的大小采样对应的图片，不就完事了吗？

Mipmap图片的保存，大概是这样的，看图：



所以为什么一般贴图的分辨率，都是2的N次方？当然是因为这么搞好处理啊。容易算mipmap啊。

讲到这里，大佬们以为自己已经无所不能，已经完事了吗？图形学换我我也能？

没那么简单。不过确实，也差不多到头了。上面的采样，中文翻译叫：双线性采样。更高级麻烦一些的，还有两个，一个叫三线性采样，一个叫各向异性过滤。

我们先来讲一下三线性采样。

我们为什么需要三线性采样？三线性采样主要解决什么问题？

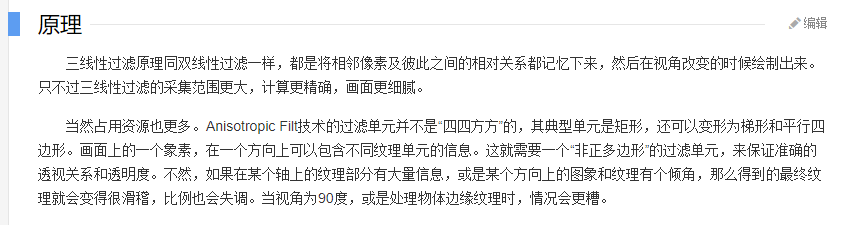
假设一个MESH比较大，用mipmap采样的时候，可能会出现一个问题：离镜头近的地方，mipmap是N，远一点的地方，是N + 1.那么会出现什么情况呢？过渡的地方会出现明显的一条线一样的不平滑。因此，三线性采样，是解决不同mipmap采样的过渡问题。

怎么解决？当然好解决啊，先采样N层mipmap的，再采样N+1层mipmap的，两个采样得到的颜色，再做一次线性插值即可。所以双线性插值的采样次数是4，而三线性采样的次数是8.

各向异性过滤。

这名字听着就蛋疼，我觉得翻译得不好，并且不要顾名思义了，没意思。

为什么需要这个东西？主要解决什么问题？我先来吐槽一下网上的主流解释。

先来看看度娘百科的解释，你看懂了吗？或许是我的天赋有限，这个解释，我如看天书！

我们再来看看度娘搜索到的其他解释，有一个解释大家各种抄袭，我估计是某人翻译过来，然后大家一通抄，现在我也抄出来。



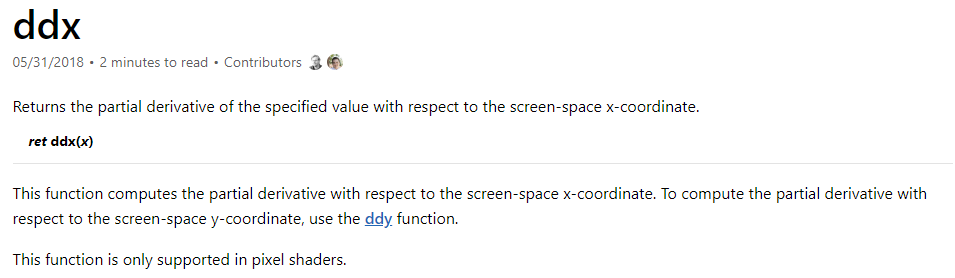
这个是网上烂大街的解释，各种blog，各种文章，几乎全是这个。里面有一句话：其实细节我也没搞清楚。我猜测翻译的哥们还是比较谦虚的。既然翻译的哥们没搞清楚，那么后续抄抄抄的大概率也是没搞懂的。那么看了这些文章，能搞懂什么呢？我就搞懂了：当一个MESH倾斜于镜头的时候，纹理采样是会有问题的，开了各向异性过滤就好了。至于为什么有问题？如何解决问题？看起来还是一无所知。

这个问题其实也困扰过我。真正完全搞懂了这个问题，是在我真正理解了如何求Mipmap level的时候。那么，渲染的时候，mipmap level是如何求的呢？我们来思考一下，假设完全屏蔽掉显卡，我们纯写算法来实现一个软光栅，那么，如何来求这个mipmap？

我想过无数种比较幼稚的方法，都没有GET到GPU的G点。但是思路应该都是没问题的。大概的思路应该是：

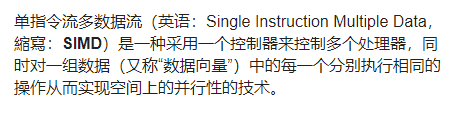
假设我在3D里渲染一个QUAD，平铺一张1024 \* 1024的图片，如果需要采样最顶层图片，那么，渲染的QUAD在屏幕上占的像素，必须大于512 \* 512。这个很好理解吧？一旦更小了，就要用更小的图片进行采样。如果这个不能理解，回头认真阅读采样部分的内容。再不能理解，直接GG。

如果采用了512 \* 512，那么两个像素的间隔U，假设连续两个像素的U间距定义为U\_Interval，那么1 / 1024 < U\_Interval < 1 / 512。所以，核心是两个连续像素的UV间隔，一旦两个连续像素的UV间隔很大，例如两个像素的间隔去到1，直接采样2 \* 2的贴图就好了，已经别无选择。

总结：得到连续两个像素的UV间隔，是计算mipmap level的核心。如何得到？有两个比较奇葩的函数，在HLSL里面叫做ddx，ddy。这两个函数，翻译叫做“分别求屏幕坐标在x方向，y方向的偏导数。”当初我看到这个，完全是一头雾水。直接看MS的说明都不管用，看图：

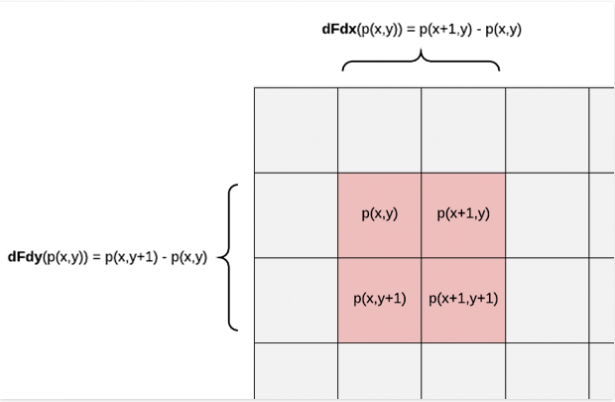
后来，混一些论坛混久了，偶尔从一些搞GPU硬件的大佬的只言片语，看到了一些消息，然后自认为理解了这个东西，不保证绝对正确。因为这个路子比较野。

首先，GPU渲染的时候，绝对不是一个像素一个像素慢慢渲染的，那样效率太低了。正确的姿势，应该是一次性渲染好多的像素。我好像看到的是32还是64？我觉得这个会不会太保守了。这里的具体数字，估计要问NVIDIA的大佬才知道了，说不定不同款的显卡都不一样的，所以不大需要深究。我也不知道。

一次性渲染多个像素，不大可能是一条指令可以做到的。这里据说用的是SIMD指令。这个SIMD指令在后续的优化章节里面我会详细介绍一下，这里只作简单介绍。

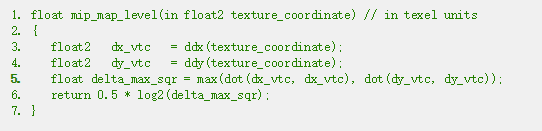
这是网上搜的，我自己懒得写了，反正一样的。大意就是：一般32位的CPU，指令集也是32位的，一条指令一次处理32位的数据。SIMD的意思是，一条指令可以处理128位的数据。也就是说，128位的数据，正常的指令是四个，现在一个就完事了。那么，为什么查看CPU指令集的时候，压根没有什么SIMD指令集呢？其实就是SSE，SSE2之类的，SIMD只是统称。

因此，假设GPU用的是SIMD指令，那么一次处理的像素是4个，数据在同一个寄存器，那么，ddx，ddy函数，讲半天其实获取的是相邻两个像素的数据，例如你传入的是当前PS的UV，那么获取到的就是跟相邻像素的UV差。如图：



本图来自网络。

理解了上图，那么大概可以推断出如何计算mipmap level了。下图是网上找的据说是计算mipmap level的函数。

这个函数不知真假，不过我觉得思路是没有问题的。为了完全看懂这个函数，我还专门去重新看了一遍指数函数，对数函数。

大概解释一下：

第一步，先去获取到当前像素跟周边像素的UV向量。其实就是跟周边UV做一个减法。

第二步，点乘其实就是平方，计算长度。说穿了，就是计算获取到的UV的长度。但是，计算向量的长度，不是应该sqrt(x^2 + y^2)这样的吗？你说得没有错。这样也是可以的。但是sqrt是一个计算量较大的操作，因此后续优化了。

第三步，取得x和y方向最长的那个UV，也就是偏移最大的那个UV。

第四步，这个是核心关键，转换一下，这个其实就是：*log2(sqrt(delta\_max\_sqr));*

所以，看到了吗，这里完美的避开了sqrt操作。

这里，难以明白的就是返回值。按照这个计算方式，返回值应该是个负数。所以这里涉及到显卡底层的实现。这块，我就真的不知道了，也没见过相关资料。有知道这块的大佬可以告诉我一下，感谢。

回到各向异性过滤，那么，明白了为什么需要各向异性过滤了吗？看图：



假设ABCD是四个屏幕上相邻的像素，在屏幕上，B-A=1，C-A=1，这个很好理解。但是回到3D，由于整个地面是倾斜的，(C.UV - A.UV) >> (B.UV - A.UV)。

明白了吗？纵向的UV坐标差远大于横向的UV坐标差。

回到之前计算mipmap level的算法，显然，这里会取最大的纵向坐标差。

所以，想当然的，这里就采样了很小的图，导致了模糊。

各向异性过滤如何解决这个问题？据说是采样临近更多的mipmap，然后求平均值来实现的。这个好像在维基百科里面有看到过，具体的实现估计做显卡的大佬最清楚了。

其他纹理

纹理其实有很多种，除了常见的2D纹理，还有1D纹理，还有3D纹理，还有CubeMap……

1D纹理没什么说的，基本不用。3D纹理主要用在一些noise之类的，我通常用来做一些序列帧动画。我用过的3D纹理，基本都是dds格式的。其他格式的是否支持这个的不知道。序列帧处理起来也超级简单，都是烂大街的东西，也没有太多可以说的。

需要重点说说的，是CubeMap。说白了就是个正方体采样，本质还是2D图片，但是区别在于UV的计算。不过不打算在这里讲，而是在阴影那一章再细讲。

纹理采样都讲完了。这里抛出一个问题：渲染一个MESH，不开mipmap，用一张小贴图，会不会比一张大贴图效率更高？

答：除非一些极端情况，例如显存耗尽。否则不会。很简单，纹理的采样其实就跟cpp里面的指针获取数据一样的。Col = Data[100]; Col = Data[10000]，这两个取值的效率绝对不会存在什么量级的差距。因此，用不同分辨率的图片，仅仅会影响显存的占用，并不会影响渲染的效率。

理解不了？放弃吧，这都理解不了，你以后怎么跟大佬们谈笑风生呢？怎么能维护世界和平呢？你太弱鸡了。

好了，纹理以及纹理采样相关的内容，就只讲到这里了。后续一些实战方向的，例如阴影图，法线贴图等等，在相关的章节里面再详细讲。下面讲材质。

材质。

刚入行的时候，听说到材质这个词，基本上都是一头雾水。很容易望文生义的误区。我一直在想，材质跟现实中的材质是不是一样的东西呢？现实中的材质，不就是什么玻璃材质，木头材质，钢铁材质诸如此类的东西？

陷入这样的误区，往往需要大量的时间拨乱反正。我曾试图在网络上找到3d材质的解释，但是遍寻网络，都没能找到一个让我廓然开朗的解释。那么，现在，我试图做出解释，能不能让人豁然开朗，绝对不敢保证。

3D材质，那是对渲染物体的数字化描述。而这个数字化描述，由于渲染函数的不同，渲染器的不同，渲染引擎的不同等等，会有极大的不同。

大概十几年前，还是固定渲染管线的年代，显卡的渲染函数是固定的，材质的参数是固定的，常见的参数有：贴图，自发光，漫反射，镜面反射……。那么，这些参数，你都可以理解为材质的参数。可以理解为一个类，这些都是这个类的成员变量，用于计算光照。

除了计算光照，材质还有其他一些参数，例如alpha混合，渲染状态设置，裁切设置等等。说穿了，材质你就理解为一个结构体，里面记录了你渲染整个mesh需要用到的数据！

那么，为什么说这些数据可能差异极大？举例：我现在渲染一个烘培好的场景，我的pixel shader里面，就一行代码：

Float4 PS(Texture, UV)

{

Return tex2D(Texture, UV);

}

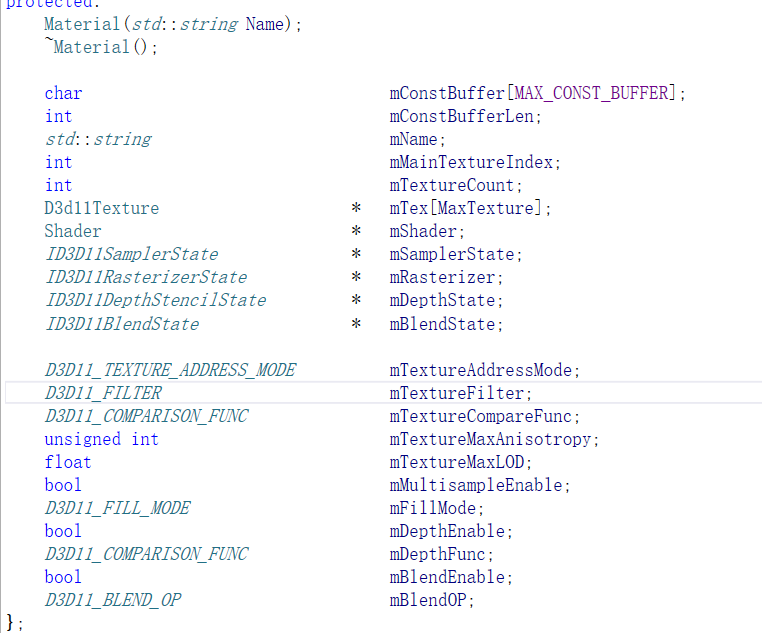
那么，你什么自发光，漫反射，镜面反射，什么都没用，全部变成了无效参数！

再如，传统的光照，用的是漫反射，镜面反射那一套算法，但是PBR光照，换了完全另外一套算法，也就是说，两种材质的参数发生了巨大的变化。另外，当你用光栅化算法的时候，跟光线跟踪算法的时候，材质的描述也可能是完全不同的两套东西。

因此，在我这里的结论：材质就是一套数字化描述Mesh渲染的数据。这套描述没有统一标准，会根据算法的不同而区别极大。

好了，材质的解释到此为止，接下来，我打算讲解一下渲染流程，加深对材质的理解。

先看看我随手写的一个极简的材质，如图：



这类极简材质，完全不需要深究，有个大概印象即可。也就是说，极简材质，主要就是一些常用的渲染参数设置，例如混合，寻址，渲染状态等等。

这个东西随便看看即可，不用当真，因为这个东西在引擎里面完全不可能这么用！在引擎里面，需求远比这个要复杂得多。说穿了，这么简单的材质，只能满足这么简单的渲染流程：

Void Render()

{

SetRenderState();

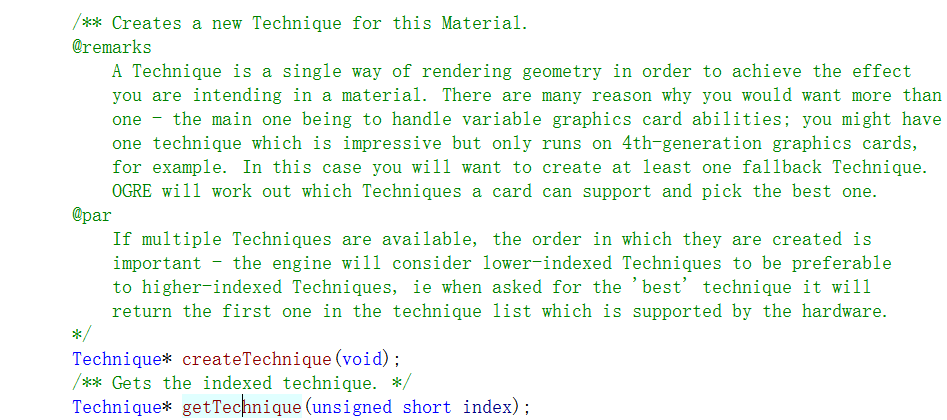
Draw();

}

SetRenderState就是设置各种渲染参数，这部分渲染参数可以保存在material结构体中，也可以保存在其他地方，看引擎的设计。

现实情况远远不是这么简单。举例：你做一套渲染，可能需要考虑高、中、低档的渲染，这种简单粗暴的设计，怎么实现得了呢？就好比好的显卡用某些参数，用某些shader，差的显卡用另外一套参数，用另外的shader。

再比如：某一个Mesh，你渲染到显示窗口的时候，跟渲染阴影图的时候，甚至跟渲染到其他RTT的时候，很可能都不能是同一套参数，不是同一套shader，怎么搞？

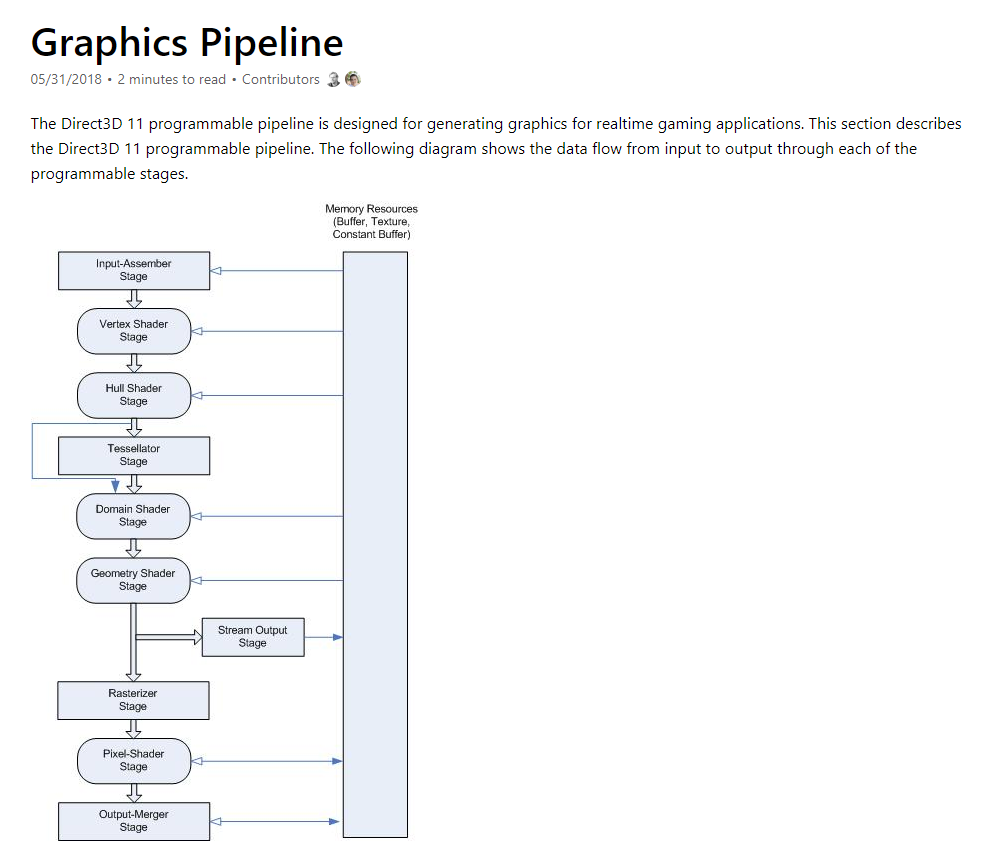
所以在OGRE里面，在材质里面，引入了另外一个东西，叫Technique。看图：

这里其实解释得很清楚了。如果看不懂英文，强烈建议学学英文。计算机搞到了一定程度，必然对英文有较高的要求，无论是看论文还是看文档。如果这个要求达不到，真的很影响个人能力的提升。

上面英文的意思跟我之前讲的差不多的，你可能有很多原因需要创建不止一个technique，可能需要匹配不同的显卡等等，这个时候，你就需要不同的technique来实现你的需求。例如你每一个material，都增加一个渲染阴影的technique，然后当你渲染阴影图的时候，全部mesh渲染的时候，切换到相应的阴影渲染technique即可。关于这一部分，我可能在后续的阴影章节里面做一个详细的描述，这里不打算深入。

除了匹配不同的渲染需求，还需要匹配同一个渲染需求的循环。这个典型的需求就是实时灯光的渲染。

这里，是时候上一张非常重要的图了：



这是dx11的渲染管线图。刚开始看不懂？没有关系，第一次看不懂，就留意一下，一次一次的看。我觉得，没有一定的基础，一次看懂不现实。

这里还没打算详细讲解，只讲解一个比较常用并且比较难以理解的东东，那就是pixel shader。这个东西翻译成像素着色器。你可以理解为一个函数，这个函数最终计算出MESH在屏幕上显示的颜色。例如：

Float4 PS(Texture, UV)

{

Return float4(1, 0, 0, 1);

}

这里，虽然传入了贴图，传入了UV坐标。但是返回值就一个红色。（Float4就是rgba四个通道，每个通道是0-1之间的浮点数）。你用上面的pixel shader，去渲染MESH，随便你各种套路，出来的就是一坨红色。

最简单的最常用的pixel shader，就是做一个简单的贴图采样的shader，大概是这样的：

Float4 PS(Texture, UV)

{

Return tex2D(Texture, UV);

}

以上函数，就是根据贴图和UV坐标，做纹理采样，取得颜色。一般在渲染UI，或者是做一个简单的demo，不考虑光照的时候，这样就可以了。

但是，现实中，这样肯定是远远不够的。我们需要实时光照，我们需要实时阴影，这个函数就该弄得复杂了。我大概写一个稍稍复杂的伪代码：

Texture BaseTexture; // 基础贴图

Texture ShadowTexture; // 阴影图

LightParam Light1; // 灯光参数

Float4 DIffuseCol; // 漫反射颜色

Float4 SpecularCol; // 镜面反射颜色

Float4 PS(float2 UV)

{

// 采样基础颜色

If( Light1.PointLight) //点光源

{

//计算光照

}

Else if(Light1.DIrectionLight) // 平行光

{

//计算光照

}

Else //探照灯

{

// 计算光照

}

// 计算阴影

// 返回最终颜色

Return FinalColor;

}

上图，就是一个稍稍复杂的光照计算。计算光照部分，一般需要用到漫反射，镜面反射作为参数进行计算。正常的计算里，往往会需要判断灯光的类型，不同的类型可能用一样的算法，也可能不一样的算法，我都见过。

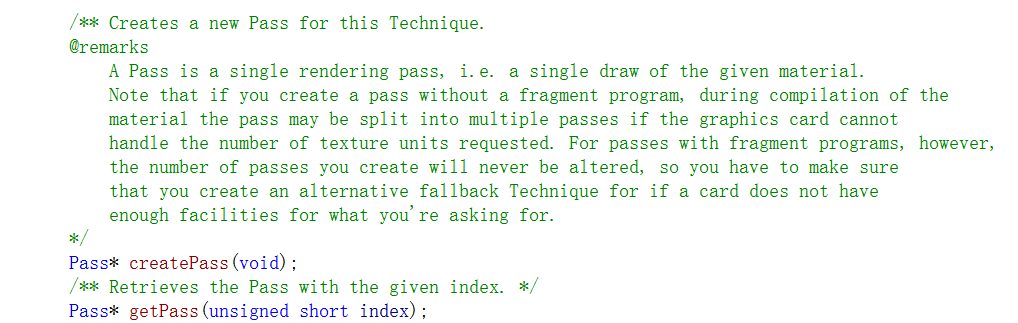
看到以上的渲染过程，新手往往会比较困惑：这不是很完美吗？有什么问题呢？另外，这个章节不是讲材质吗？这个跟材质有什么关系？

首先要理解：材质是服务于渲染流程的你渲染过程中需要什么参数，那么我材质保存什么参数，传入进来。所以，渲染流程对于材质来说，才是至关重要的东东。

那么，以上的渲染过程，有什么问题呢？问题非常多，现实远远比这个复杂得多。至少，有两个可以优化的问题。

1、shader里面使用了if else这样的跳转。大概十多年前，shader里面是几乎不推荐使用这类跳转的，因为GPU的流水线，一旦使用了一次跳转，这个渲染的效率将降低一半！但是，现在几乎所有复杂点的shader，都很可能有不止一个跳转了，这是大势所趋，也是无可奈何之事。现代GPU是否有对这个跳转做了新的处理，我没有仔细研究过，但是，按照流水线的理论，能不用就不用这才是最正确的。

1. 上面的过程中，只有一个灯。如果场景中有N个灯呢？怎么处理？你能一次一次调用这个渲染吗？答案是：不能。你渲染得到的颜色，不可能通过这样叠加的形式来实现。

这些问题，不同的引擎有不同的处理方式。例如OGRE里面，会把一次渲染，定义成一个pass。一个technique可以包含N个pass，一个一个渲染，渲染的结果可以相乘，可以相加，可以混合等等。也可以一个pass渲染多次。看图：

所以多个灯光的时候可以怎么干呢？可以先把不需要计算光照的部分，抽离成一个单独的pass，首先渲染这个pass。然后每一个灯，就渲染一次光照pass，把最终的结果叠加。大概是这样子的：

// 以下是GPU的shader

Float4 PS\_NoLighting()

{

}

Float4 PS\_Lighting()

{

}

// 以下为CPU的渲染流程

Void Render()

{

RenderNoLighting();

For (int i = 0; i < LightsEffectThisMesh; i++)

{

RenderWithLighting(Light[i]);

}

}

注意，这里灯光是要做裁切的，意思是要先遍历一遍所有的灯，计算某个灯光是否照到这个Mesh。照不到的话不参与计算，这是基本的优化。

以上是唯一方案吗？当然不是，还有其他一大堆方案，我这里再介绍一些比较常用的。有的做法，会把所有的光照都在一个shader里面完成，一次性传入N盏灯。大概是这样的

Light Lights[16];

Int LightCounts;

Float4 PS(Texture, UV)

{

// 非光照计算

For(int i = 0; i < LightCounts; i++)

{

// 计算每一盏灯的光照

}

}

哪一种方案是最优的？我认为，没有哪一种能碾压另外一种。一切看你的需求以及你的选择。第一种，架构比较清晰，扩展性更好一些，但是效率明显不如第二种高。

第二种扩展性没那么好，而且早期受限于硬件，灯光数量有限制。

有朋友可能会问：现在主流的引擎是哪一种？从渲染架构来说，现在其实两种都不主流了，现在主流的是延迟渲染，forward+这类的，只是，发展的过程都是一点一点的，一口吃不成一个胖子。我认为，一点一点的理解这些过程，有助于理解更深入的东西。至于延迟渲染跟forward+，以后的章节可能会介绍。其实这类资料有点烂大街了，也可以自己去找找看。

好了，关于材质的讲解，不打算讲解太多了。本来的计划，是有打算讲解一下PBR材质的，无奈这东西都烂大街了，一般人只要熟悉这个渲染过程，就是CTRL C + CTRL V的过程，我想了又想，也不觉得自己能讲出花来，那么还是放弃吧。

回顾一下，材质其实是记录一些渲染参数，渲染流程等等。可以自己写一个最简单的材质，只需要记录一次最简单的渲染需要用到的参数即可，便于熟悉这个过程。而现实的引擎，材质系统一般都设计得非常复杂，这就不是新手的概念了。到了一定程度，可以去看看一下一些知名引擎的材质系统的设计。