本文欢迎转载，但是请著名出处：<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。

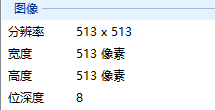
那么，既然这样，那么为什么纹理需要保存跟图片一样的格式呢？都用JPG，PNG，DDS之类的格式来保存呢？这是因为，流水线贯穿于整个GPU的始终。

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。

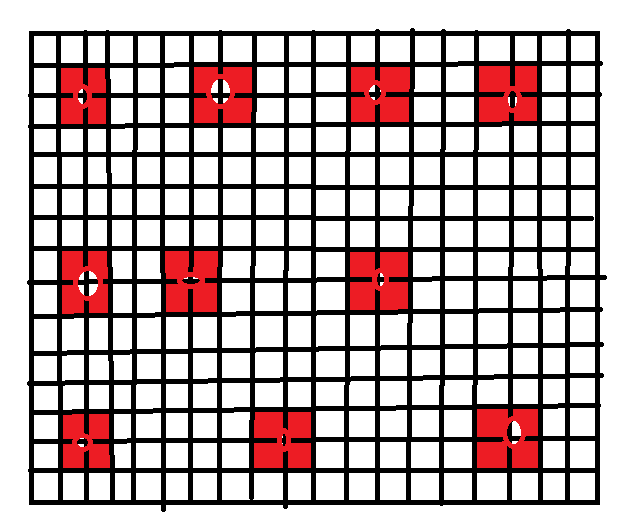
假设一张纹理，分辨率是1024 \* 1024，现在渲染一个点：UV坐标是（0.3， 0.4），那么，我们可以这样算（这里只讨论最常用的线性采样，最近点采样更加简单）：

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？

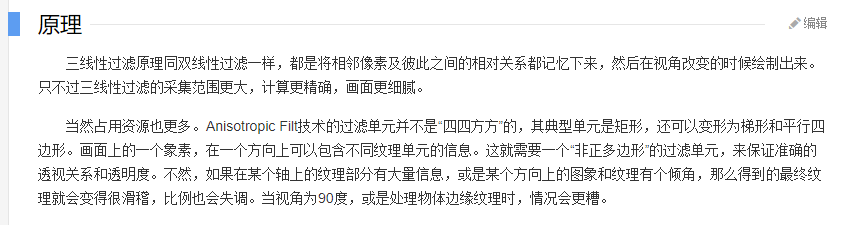
假设一个MESH比较大，用mipmap采样的时候，可能会出现一个问题：离镜头近的地方，mipmap是N，远一点的地方，是N + 1.那么会出现什么情况呢？过渡的地方会出现明显的一条线一样的不平滑。因此，三线性采样，是解决不同mipmap采样的过渡问题。

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。



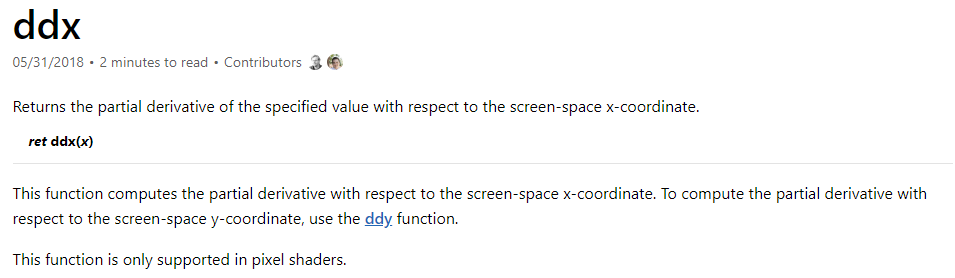
这个是网上烂大街的解释，各种blog，各种文章，几乎全是这个。里面有一句话：其实细节我也没搞清楚。我猜测翻译的哥们还是比较谦虚的。既然翻译的哥们没搞清楚，那么后续抄抄抄的大概率也是没搞懂的。那么看了这些文章，能搞懂什么呢？我就搞懂了：当一个MESH倾斜于镜头的时候，纹理采样是会有问题的，开了各向异性过滤就好了。至于为什么有问题？如何解决问题？看起来还是一无所知。

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：

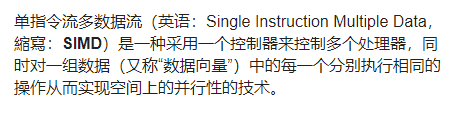
假设我在3D里渲染一个QUAD，平铺一张1024 \* 1024的图片，如果需要采样最顶层图片，那么，渲染的QUAD在屏幕上占的像素，必须大于512 \* 512。这个很好理解吧？一旦更小了，就要用更小的图片进行采样。如果这个不能理解，回头认真阅读采样部分的内容。再不能理解，直接GG。

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：

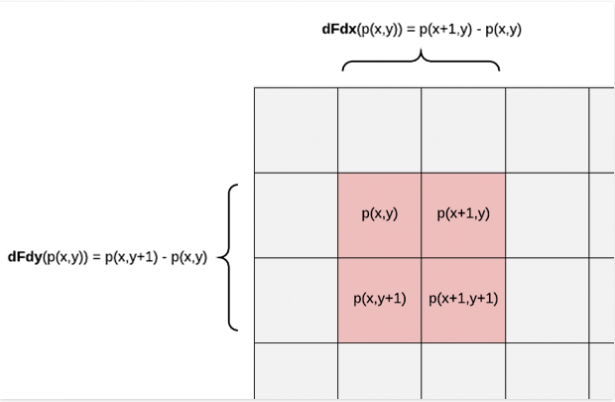
后来，混一些论坛混久了，偶尔从一些搞GPU硬件的大佬的只言片语，看到了一些消息，然后自认为理解了这个东西，不保证绝对正确。因为这个路子比较野。

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。

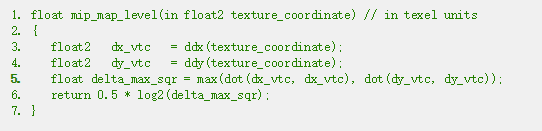
这是网上搜的，我自己懒得写了，反正一样的。大意就是：一般32位的CPU，指令集也是32位的，一条指令一次处理32位的数据。SIMD的意思是，一条指令可以处理128位的数据。也就是说，128位的数据，正常的指令是四个，现在一个就完事了。那么，为什么查看CPU指令集的时候，压根没有什么SIMD指令集呢？其实就是SSE，SSE2之类的，SIMD只是统称。

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。