阴影

英文叫shadow，所以看阴影的shader代码，往往会有这个单词。看到这个单词，大概率就是阴影相关的shader了。

本章，我大概会以伪代码+真实自己之前写的代码来讲解。我查了一下，我的代码全部写于2011年，时间已经比较久远。所以，说不定有些细节我都记不清楚了。不过没关系，我确定这些代码是能跑并且666的，这毕竟是我之前做一个装修平台的部分核心代码。在当时来说，还是很不错的，比unity当初默认的简单效果好得多了。

另外，由于模板阴影是已经淘汰了的技术，本章也不打算讲。想了解的，网上资料应该也已经烂大街了。

就我看来，阴影技术，延迟渲染技术，属于图形学的分水岭。能深入理解这些技术的，在我看来，好歹是比较理解了渲染架构，渲染流程，显卡等等，脱离了一般API程序员的行列。

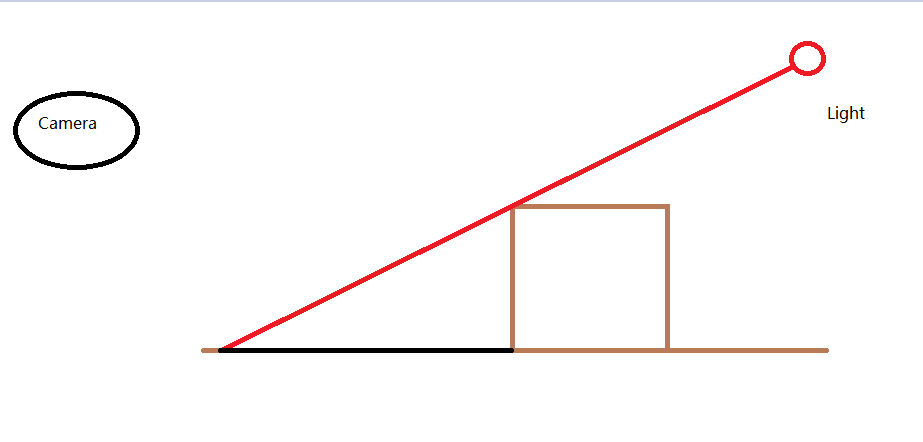
一般来说，我看简历上提到自己熟悉图形学的，我会先问一些简单的问题，如果都能回答得可以，那么我会问阴影相关的。

第一个问题一般是：3D里的阴影是如何产生的？一般人这个时候，都会提到什么z-pass算法，什么z-failed算法。然后再问一下实操流程，一脸懵逼的是学院派的，估计看过书，但是从来没有实操过，也没有深入理解。如果回答拖个相机，生成阴影图，再如何如何计算阴影的，那理解就深入多了，可能实操过的。

这个时候，我一般会问：那么点光源怎么拖镜头？点光源FOV你怎么设置？平行光呢，你又放哪里？

到了这一步，我面试过一百几十人总是有的，没有人回答出来过。不知道是这个问题太刁钻还是我们小公司，来应聘的人水平不够。也可能是现在做图形学的不多了吧。

先来简单介绍一下光照的原理以及实现。先看简单的，Spot Light，看图：



看图，黑色部分是阴影部分。这个时候，无论Camera在哪里，阴影还是一成不变的，所以，可想而知，阴影跟Camera的位置无关，只跟遮挡物、跟光源位置有关。

具体实操怎么搞？一会我先用CPU的伪代码，解释好过程，再上GPU的伪代码。很多人一开始看不明白，是因为你看到的都是GPU部分代码，这部分很多都是千锤百炼，各种优化过的，让人很难看懂。

1. 先在Spot Light同样的位置，同样的参数，放一个Camera。参数跟Spot Light保持一致。这一步，叫Render To Texture。但是，渲染的不是颜色，而是深度图。目的，是保存当前灯光方向最近的阴影。这一步，在CPU里可以这样写：

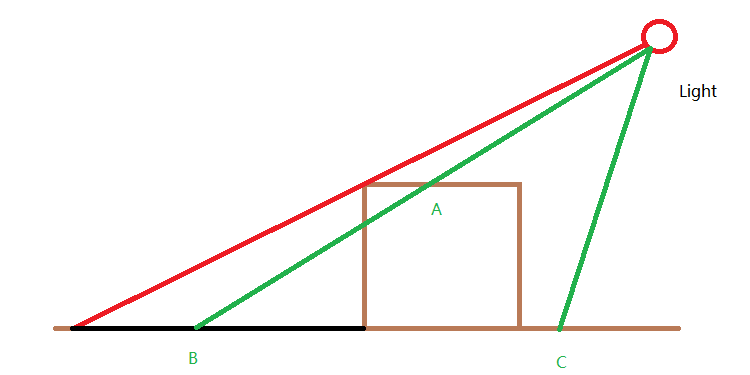
Float4 worldPos = mul(WorldMatrix, Pos);

Float Distance = (worldPos - LightPos).length;

第一个是把当前位置转换伪世界坐标。第二个是计算当前世界坐标到灯光世界坐标的距离。这部分只是简单的向量运算，如果都理解不了，这个章节真的没法看了，图形学也是搞不下去。

第一步执行完之后，你会得到一张贴图，这张帖图记录了距离灯光位置最近的所有位置。这张图，我们一般叫阴影图，ShadowMap。

1. 渲染实际MainCamera，计算当前位置到灯光位置的距离，然后取当前位置距离灯光最近的位置，做比较，大于最近距离，那么证明在阴影之下，需要画黑。看图：



当渲染B点的时候，去取距离灯光的最近点，得到的距离是A到Light的距离，而B到Light的距离，大于A到Light的距离，那么，B点就该渲染阴影。而渲染C点，取得到的距离，都是C到Light的距离，那么C点无阴影。

这么解释，我觉得比什么z-pass，z-failed之类的，会不会简单易懂一点？

这里，CPU的伪代码大概是这样子：

Float depth = (worldPos - LightPos).length;

Float depthShadowMap = tex2D(ShadowMap, UVShadow);

If(depth > depthShadowMap)

{

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

}

Else

{

Return tex2D(Texture, UVTexture);

}

这样的话，如果是阴影区域，会返回一个纯黑的阴影。如果非阴影区域，采样图片得到简单的无光照颜色。

以上代码，我们可以轻易看出：

1. 需要两套shader，第一套，是生成阴影图的shader。第二套，是渲染阴影的Shader。
2. 每增加一个实时阴影，整个场景等于重新多渲染了一遍（生成阴影图），效率消耗极大。

以上算法，应该还是比较简单易懂的。如果这都看不懂，后面的压根没法看。但是，以上算法，只是一个最简单的雏形，还有非常多的细节需要搞的。例如：

1. 浮点数精度会不会导致这个距离判断出现问题，导致没有阴影的表面也上了阴影？
2. 这个深度可以这样直接保存吗？
3. 如何得到当前点在ShadowMap上的UV坐标？得不到的话怎么正确得到当前阴影图上记录的最小距离？
4. 如何阴影抗锯齿？
5. 如何让阴影不那么黑？
6. 多灯光的时候，阴影之间互相的干扰怎么办？

………………

实际使用中，阴影的问题可以复杂到爆炸。我先来解释一下为什么阴影的GPU代码比CPU的难以看懂那么多。

第一个问题：GPU的阴影图，只能保存0-1之间的数值，而不能直接保存计算得到的距离。为什么这样呢？不伪什么，只是为了更好的控制精度罢了。否则，如果是纯浮点数的精度，整数部分会造成极大的浪费。要理解这一点，必须对浮点数的精度以及浮点数的储存有深刻的认识。

基于这个原因，第一个shader那里，直接计算depth储存下来的代码，在GPU里是行不通的。

所以一般可以这么干：

Float4 worldPos = mul(WorldMatrix, Pos);

Float Distance = (worldPos - LightPos).length;

Return Distance / CameraFar;

最终保存的depth，可以除以镜头的最远距离，这样，就能保证数值在0-1之间了。这是其中一种方案。但是，这里的门道就非常多。以前，由于比较菜，对图形学的理解不够，很多门道都不好看明白。

例如这个门道：

Float4 worldPos = mul(WorldMatrix, Pos);

Float Distance = (worldPos - LightPos).length;

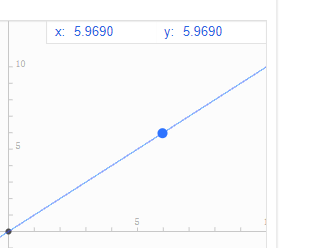
Float d = Distance / CameraFar;

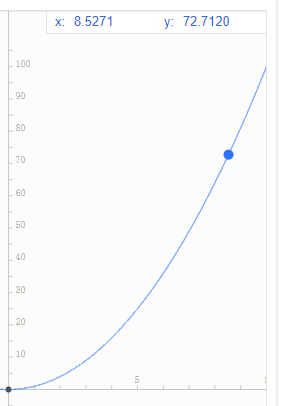
Return d \* d;

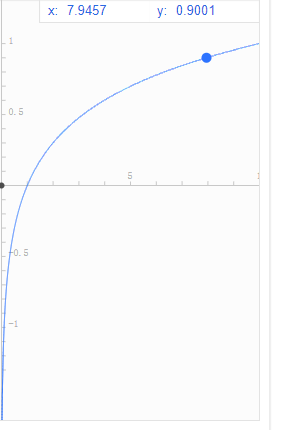
我当初就纳闷了，你直接返回d不就完事了吗，为什么需要返回d\*d？有区别吗？

这就是有限的精度，你主要用于哪里的问题。这里我简单解释一下，能明白的就明白，不能明白的说不定三五年后自己就明白了，也可能一辈子不需要明白，只需要明白大概。

看图：

这是函数y = x的曲线。

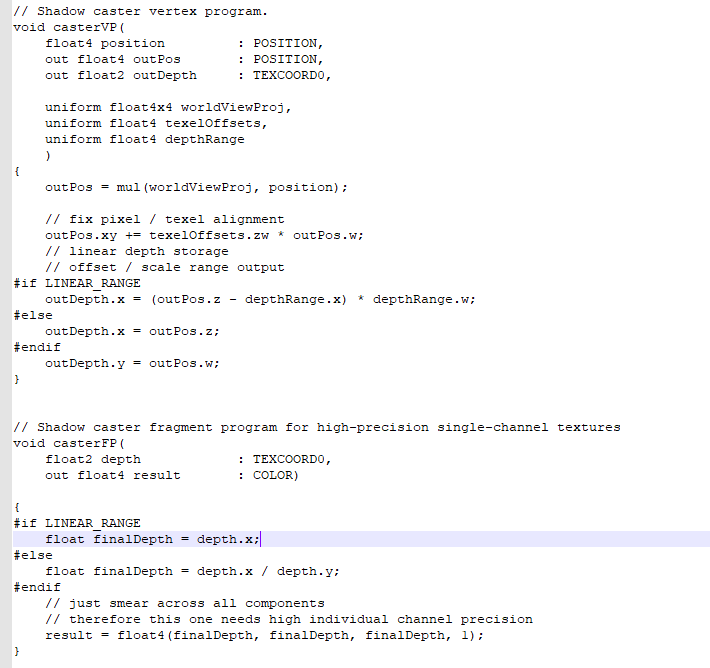
这是函数y = x^2的曲线。

这是y = lg（x）函数的曲线

这里，线性的曲线的意思，是精度是平均的，远距离的精度，跟近距离的精度，是一样多的。当场景比较普通的时候，用线性的精度会比较好。但是，如果渲染的物体距离灯光都比较远，那么，我们更应该把精度往远的地方倾斜，这个时候，用log函数是比较常见的处理方式。而如果大部分物体距离灯光都比较近，用y = x^2就可以理解了。

这么解释，明白就是明白，不明白就是不明白，我也不知道怎么解释才能更清楚了。所以，由于骚套路比较多，导致了很多shader算法非常不好看懂。

再看看这个骚套路（这个其实还不算骚套路，只是常见的套路，骚只是相对CPU计算来说）：



这个是去OGRE的资源里面Copy下来的，按道理是不会有什么错的。但是那个depthRange我没有仔细去看怎么来的。

看看非LINEAR\_RANGE的情况下，其实直接返回的是透视投影变换那个Z。注意，这个Z，大概率是非线性的，关键是看你的投影变换矩阵是如何构建的。我记得我专门看过文档，介绍了DX的投影变换矩阵的构建，那个Depth的构建非一般的复杂，不是简单的线性。

这么做，就是简化计算，提升效率。因为本来这个矩阵乘法你就是要做的，这样可以直接得到，而不需要自己再算一遍距离，再除以镜头最大距离。

阴影图的生成，还有其他一些套路，不过套路还不算多，阴影的绘制，套路要多得多。我们来一个一个的看。

第一个新的概念，这个一般中文翻译做“贴图矩阵”，英文一般是这样的：TextureViewProjection，又或者TextureViewProjectionWorld。看图：



以前刚搞图形学的时候，看到这个就懵逼了，这是什么鬼？贴图为什么还有个矩阵？其实没那么复杂，这个就是那个阴影图的矩阵。一切Render Target，渲染到纹理，这个纹理都有这个生成矩阵。这个矩阵你可以带wordl属性，也可以不带，反正这个world变换在哪里做都是差不多的。这个有什么用呢？做过软光栅的都知道，这个变换之后，你得到的float4的xy分量，其实就是这个点在这个图的uv坐标。有那么简单？其实要稍稍复杂一点，有可能需要做坐标变换。例如屏幕坐标系的变换，贴图坐标系的变换等等，这个看你矩阵的最终计算。大概率在很多代码里，你看到的是xy/w即可得到你需要的uv坐标。

第二个新的概念：软阴影。

这个是相对于硬阴影来说的。模板阴影就是硬阴影。

软阴影的意思，是阴影的亮度，是可调的，阴影是可以做各种优化的，例如边缘是可以虚化的，做抗锯齿的，阴影是渐变的。

要做到这些，相当的不容易。

回到之前CPU接收阴影的代码：

Float depth = (worldPos - LightPos).length;

Float depthShadowMap = tex2D(ShadowMap, UVShadow);

If(depth > depthShadowMap)

{

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

}

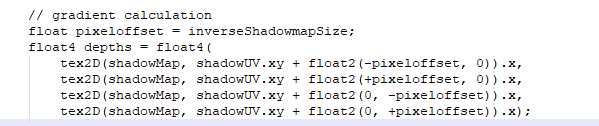
Else

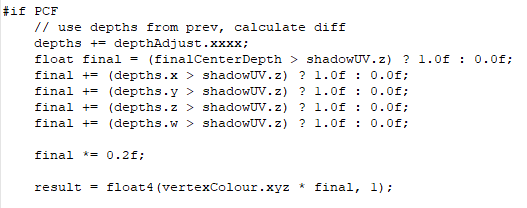
{

Return tex2D(Texture, UVTexture);

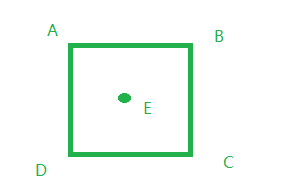
}

这就是个纯黑的阴影，是个硬阴影。我们不能用这种纯黑的阴影，效果太糟糕，我们需要边缘虚化的软阴影，这个专业术语叫做阴影的PCF。怎么实现呢？其实就是一个简单的模糊差不多的算法。看图：



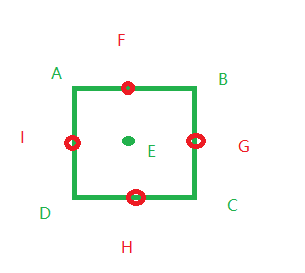


以上代码摘自OGRE。这个是非常简单的算法，其实就是采样了中心点，再采样了四个顶点，求平均值差不多。如图：



采样了A、B、C、D、E五个点，加权求平均。这样，边缘地带，假设只有B、C两个点是有阴影的，AED是没有阴影的，算起来，这个点就会有淡淡的阴影。

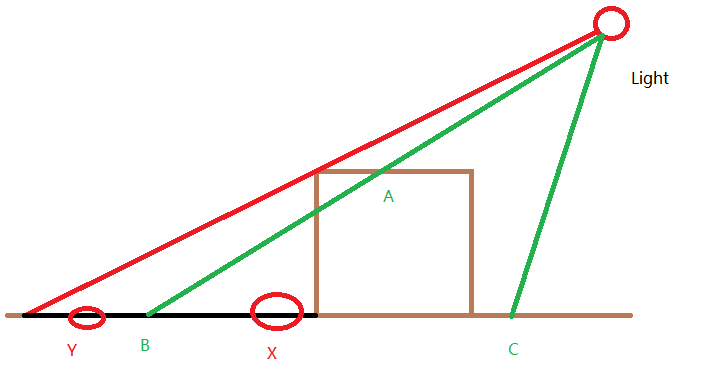
PCF的方法其实很多，我尝试过3 \* 3的，也尝试过4 \* 4的，感觉差别不大，反正不好看出来。3 \* 3的，其实就是多采样这几个点：



多算了F、G、H、I四个点。

至于怎么算隔壁点的UV坐标？我估计在其他地方都解释过无数遍了，这里再解释一遍：uv坐标，直接(+ -) 1 / textureSize即可。如果这个套路都不熟悉，图形学压根混不下去。

这个，仅仅是实现了边缘的虚化，你可能还需要其他的效果。例如：



同是阴影，你是否需要考虑X点的阴影，跟Y点的阴影是否有所不同？

我不知道别人有没有考虑过这个问题。我是考虑过的。

我们可以来做这样一个尝试：

假设从shadow map获取到的深度，是depth / camera\_far，那么，现在算到灯的距离，是current\_depth / camera\_far。我们可以做一个简单的计算：

Float factor = (depth / camera\_far) / (current\_depth / camera\_far);

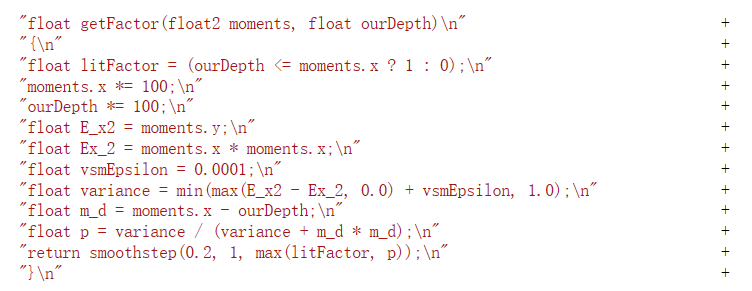
然后用个clamp函数，把factor 确定在0，1之间。

然后我们做一个阴影的插值，这个时候，最好不要用lerp，因为线性插值其实效果不是那么好，我们可以用smoothstep，平滑插值。由于阴影不能太亮，也不能太黑，所以我们可以限定一个插值范围，例如0.2-0.4之间。那么大概就是：

Return smoothstep(0.2, 0.4, factor);

这样出来的效果如何？只能说，just so so吧。后来我又觉得这个是不是因为线性的原因导致效果不大理想，改成了log函数，也就是说，factor先log一下，让变化更小。效果好像好了一点点，反正肉眼不好看出来。

再后来，我去网上找了一个版本，抄抄改改，用上了。其实效果也感觉跟我自己写的差不多，但是我当初水平有限，总觉得别人写的就是好的，就用了别人的。代码在这里：



这个计算就复杂了很多，不过现在看来，也是一些取巧的法子，应该也没有什么理论依据，只是单纯的希望算出来的阴影不那么的死板，生动一些。

上图中，moments就是阴影图采样到的d= depth/range。有兴趣的，可以仔细看看这个计算过程，其实也挺好看懂的。我曾经尝试过不断修改moments的一些值，例如弄成d^2，各种各样的尝试我记得都做过，然后发现效果没什么太大区别。所以，我的结论，这块只是让阴影生动一些，没有那么的死板。

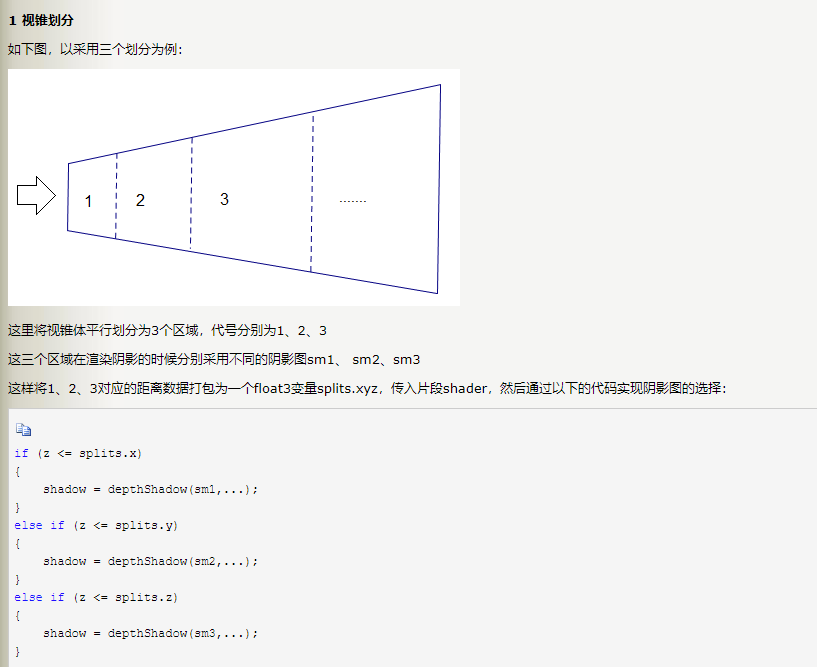
好了，聚光灯的阴影计算就讲到这里。理解了上面部分，其实已经理解了大部分。后续，讲讲点光源以及平行光。点光源很复杂，相对来说，平行光要简单一些。

平行光阴影，其实就是模拟的聚光灯，只是让灯光的范围更大。例如，可以把镜头FOV设置成120度，可以把镜头放远一些，这样，一个场景就能笼罩了一大部分。也有其他一些做法，例如透视投影改成正交投影，反正就是作假，你觉得怎么样效果好就怎么来。

人物移动的时候怎么办呢？镜头跟着移动就完事了。

场景很大，希望看到远处的阴影怎么办？可以用知名的PSSM。

PSSM阴影的原理也挺简单，估计现在资料早就烂大街了。还是大概讲一下吧，看图：



上图来自网络。这个基本已经讲述得比较清晰了，直接抄。

说穿了，就是分开N张阴影图（原论文我记得是三张）。先渲染三个阴影图，然后在shader里面，根据不同的距离，采样不同的阴影图。难点在于三个阴影图Camera参数的计算。这里，网上资料太多了，我再去抄来充数，好像说不过去。

PSSM阴影弥补了大场景阴影细节可能出现的不足。但是，三个甚至更多的阴影图生成，大大增加了渲染消耗，因此，这种阴影方案需要谨慎。小场景，其实我觉得没什么必要。

好了，平行光阴影就不讲了，接下来重点讲讲更加复杂的点光源。

点光源的阴影图如何生成？第一游戏开源引擎OGRE的处理方案，都让人啼笑皆非。看代码说话：



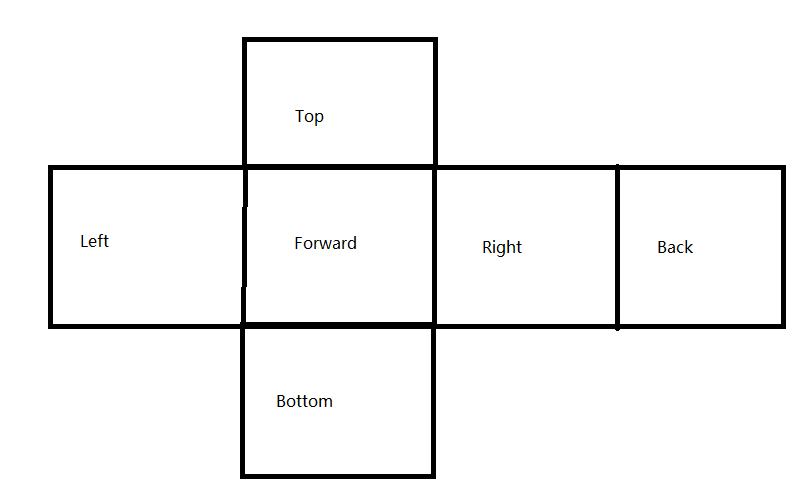
以上代码摘自OGRE1.81。

通俗易懂，就是用个120度的聚光灯，作假。这当然是不行的。当年我做家装平台，用的OGRE，发现了这个大BUG，无语凝噎。然后，自己开始修改了引擎，自己实现了一整套点光源的阴影。

点光源由于是一个圆形的全范围光照，不能用普通的阴影图，而是要用到CubeMap，大概就是这个：



其实就是6张正方形的图片这样拼起来：



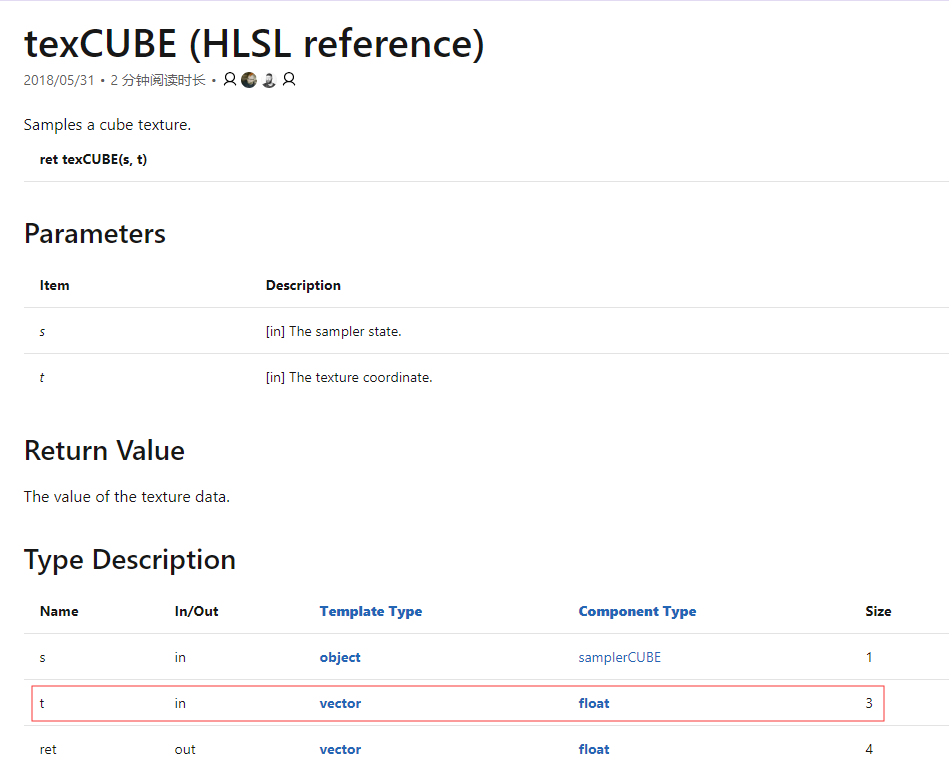
可以简单的理解为一个正方体的包围盒，跟天空盒那个是一样的。区别在于灯光是生成这个阴影图，再采样这个阴影图。

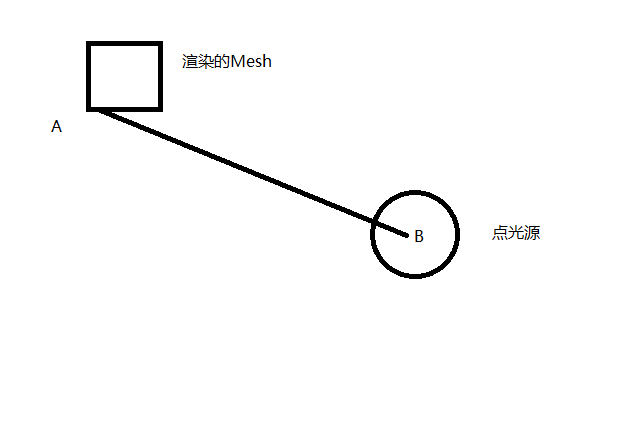
这个阴影图如何生成的？没错，就是用6个Camera！前后左右四个，顶上一个，底下一个。所以，点光源的阴影图，等于把场景多渲染了6遍，不做优化压根没法用。

那么，是怎么优化的呢？游戏里面其实用到点光源的时候甚少，而家装软件里面，我自己的优化方案是：灯光创建、移动、物体移动、旋转等一些有限的操作之后，阴影图刷新一次即可。也就是说，不需要每帧渲染阴影图，只需要场景变动的时候，刷新一次。这样，除了显存占用大一点，其他还是可以承受的，不至于跑不动。

CubeMap的采样，是怎么实现的呢？又是怎么样优化的呢？这部分复杂了很多。在资料非常贫乏的蛮荒年代，我光是查这样的资料，理解这样的算法，就花费了大量的时间。第一个，首先讲讲CubeMap的采样。

我们知道，一般图片的采样，其实就是根据UV坐标，去取某个像素点的颜色。其中，会稍稍复杂一些的是双线性采样，三线性采样，mipmap这样的东东。但是，在3D里面，这些东西大部分都是黑盒的，一般人只需要知道传入个图片，传入个UV就能得到颜色了。而CubeMap是怎么采样的呢？这个用的是空间三维采样。看看这个函数：



核心就在t这个参数，这个是个什么东西？其实，这个就是一个空间向量，这个空间向量是以Light Position为原点，到当前要采样的点的世界坐标的向量。

如图：你要算的，其实就是normalize(A - B);

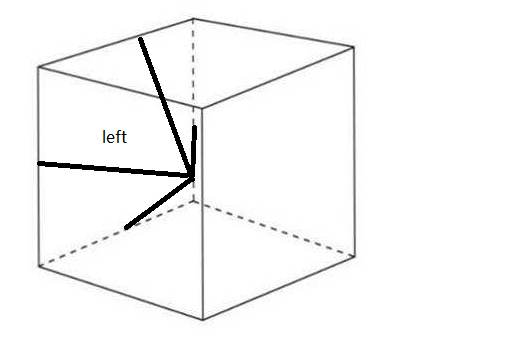
问题来了，这个是如何采样CubeMap纹理的呢？其实，主要是根据这个向量，算出来纹理的UV坐标即可。

第一步，回到之前的CubeMap图，其实是由6张正方形的图组成，组成规律是不变的。根据这向量，先计算在哪一张图（Left、Forward、Right、Back、Top、Bottom）。我还专门写过这块的CPU算法，具体不好找到了，大概是这样的不会错。

我以Left的计算为例：

1. 由于向量是normalize过的，所以x^2 + y^2 + z^1 = 1;
2. 假设x为负，才可能是left，正的话不可能。这个只需要仔细想一下即可知道。
3. 先判断了x为负，那么就需要判断y跟z。我没记错的话，是判断向量跟x轴负方向的向量求夹角，点乘就可以，计算角度，角度在范围内的，属于left采样。
4. 计算得到是left采样之后，再根据CubeMap图，实际计算UV坐标。没记错的话，UV坐标用y、z分量做一个简单的计算即可得到。

大概如下图所示：

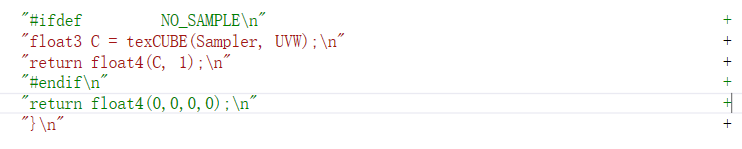


如果非要详细算法的话，可以给我留言。我得回家找找源码，太久远了。不过应该能找到的。

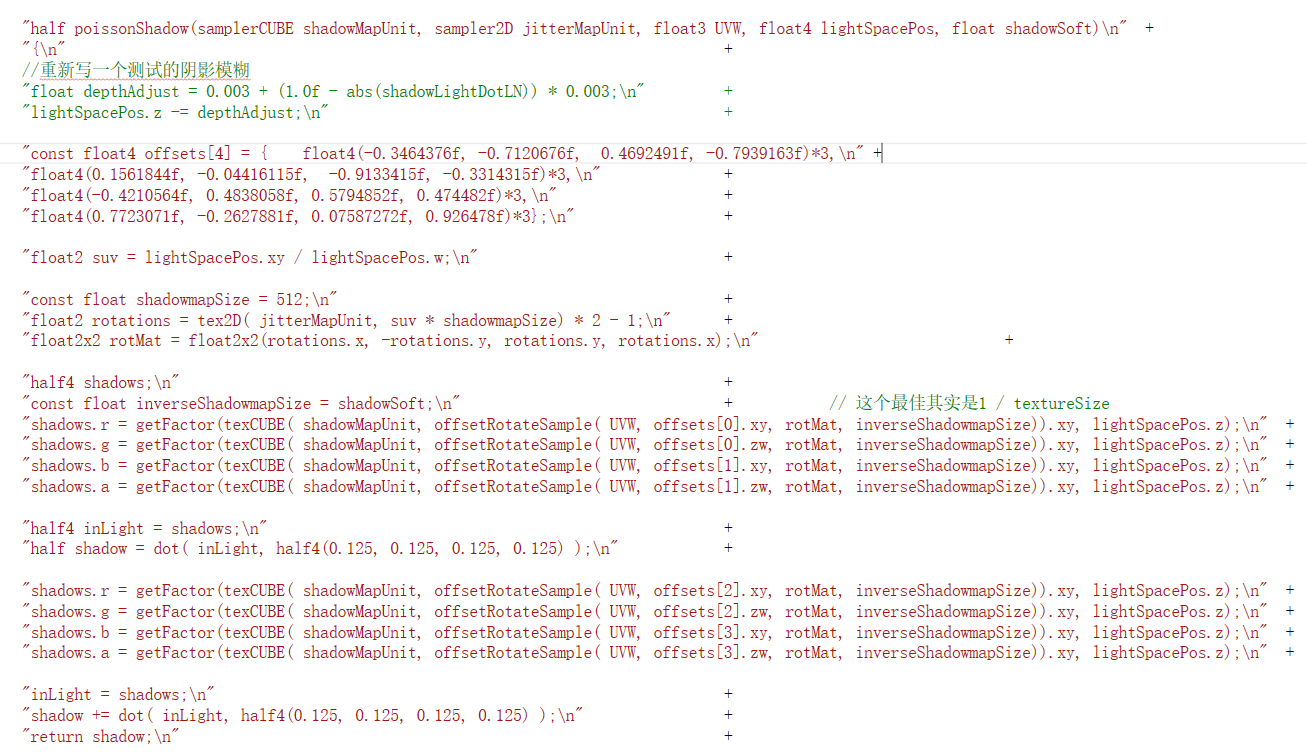
算法的核心，是根据向量，算出来是在六个面的某一个，再确定在这个面的UV坐标！

好了，这个技术细节，绝大多数人是不会的，但是不会影响到阴影的采样。因为这部分在HLSL里面是黑盒，你都不需要知道实现细节的，一个函数搞定。只是在其他一些时候，例如全景图算法的研究，不熟悉这个就坑逼了。还有，CubeMap阴影的PCF，不熟悉这个原理，也没法做，或者只能根据感觉朦朦胧胧的做。

看看点光源阴影的PCF。先发一个不做PCF的，简简单单一个采样即可：

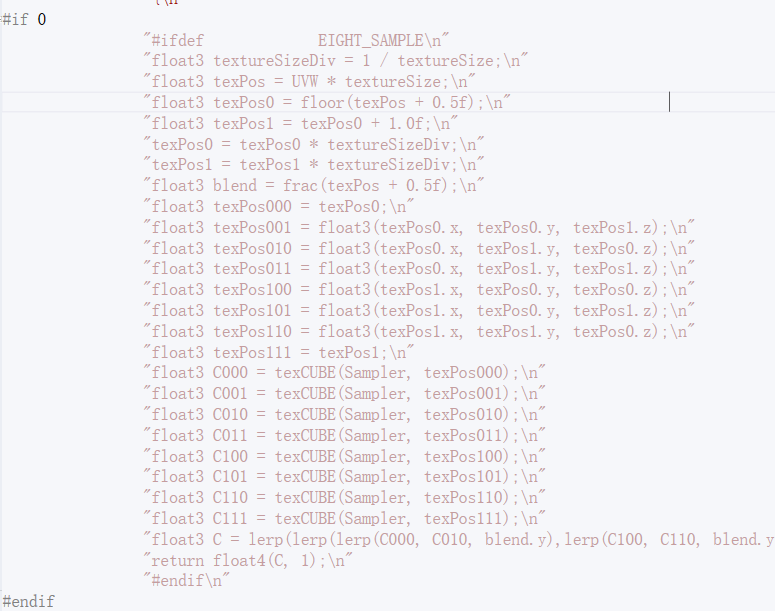


但是，这么简单的话，效果不怎么好，边缘锯齿比较严重。我折腾了很久，弄出来一个算法，这个算法一部分是抄的，一部分是自己写的。看图：



老实说，这个算法还是比较复杂的，一般的人，没有一定的图形学基础，很不好看懂。我大概解释一下。

理解上面这个之前，先理解一下一个正常的PCF采样，看图：



这个，是周边采样8次做了个加权求平均。但是，这周边8个点，如何来界定呢？

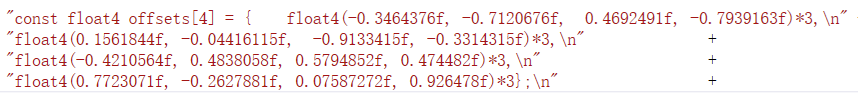
上面的算法，是求得一个8个顶点的正方体的两个斜点，算出来正方体的8个顶点，加权求平均。我最开始采用这个算法，发现效果一般般，基本没什么用，跟采样一次没什么大区别。后来各种折腾，各种改，才用了上面那个最终算法。

用了最终的算法之后，阴影边缘跟UE4那个一样的，边缘模糊，但是有颗粒状。



我的阴影也是这样边缘模糊，但是再拉近的话，颗粒状会比较明显。

核心有两部分，一部分是旋转矩阵，从一个图片获得。图片是一张奇奇怪怪，花花绿绿的图片。另外一个关键是一个叫做offsets的东东。这个东东默认赋予了一些奇奇怪怪的数字，让人完全看不懂。



图形学看得越多。越容易看到这类东西，例如有随机数的生成函数，有随机贴图，有noise图，noise生成的时候还有什么泰勒公式，还可以用傅里叶变换（FFT）来生成。经常能看到一些奇奇怪怪的参数。有的参数，例如ShaderToy上面的，很多是自己调好的，没有数学依据的。有的是有数学依据的。如果非要深究这些的原理，貌似有一本书，是专门讲图形学的各种数学的。

以上阴影这部分，我记得是有数学依据的。我印象中，应该是一种模糊算法的参数？我们知道，模糊算法，最简单的，就是周边几个像素，加权求平均。但是还有各式各样有效的模糊算法。我记得我仔细查过，这是某个模糊算法的应用？时间太久远了，已经记不清楚了，也不能保证这个理解是必然正确的。但是有一点可以肯定，做多重采样的意义，无非是边缘模糊，让阴影更加自然，变来变去，都是采样UV的细微变动，这个大方向是不会错的。

如果上述理解错了，有知道的请留言指正。

好了，关于点光源的阴影，就讲到这里。现实中，这类东西很少有人做了，一个引擎打天下，才是现在的常态。但是，我还是把各种各样的细节，都理解了一遍。也许可能一辈子都用不上，我还是觉得，知识是一点一点积累的，自信也是一点一点积累的。当所有这些细节，你都撸过一遍之后，其他知识点，你都能够轻松的handle。开头很难，但是后面会容易得多。