**Shadowmap篇：**

不错的文章：<https://zhuanlan.zhihu.com/p/144025113>

Unity的阴影，close fit不知道，stable fit是球型包围盒

**如何产生shadowmap：**

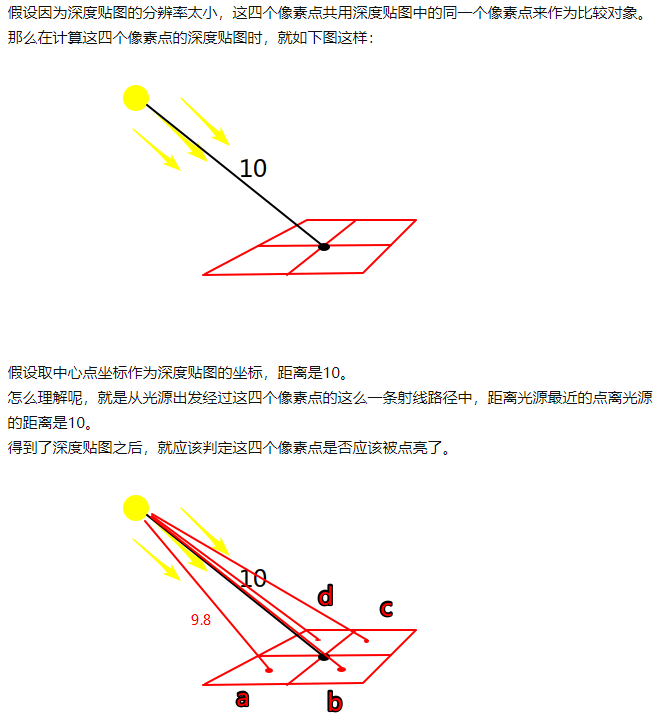
\*屏幕空间shadow的缺点，不能处理半透明阴影，最后深度图不会写入深度。

\*搞一个光源位置的相机设置为正交投影，让相机在灯光的位置以灯光的朝向去做投影，将得到的深度图存起来。

\*（receiveShadow）场景中所有物体正常进行mvp变换，要专门搞出来一个float4在vert中只进行顶点的obj->world变换，然后在frag中，使用光源位置相机的v矩阵，将点变换到和光源相机同一个空间下。对比得到的深度值和存储的深度值是不是一样，根据谁大谁小判断是否在阴影中。（注意存储的深度图要变换到视空间深度，因为recshadow中也是用的视空间的深度，总之保持深度在同一个空间下）

**如何产生并解决shadowAcne：**

\*产生是因为阴影图分辨率不够，可能多个像素（这里假设有四个）的采样都在同一个纹素上，假设这个纹素中存的深度是0.6，假设刚好这四个像素的深度分别为0.59,0.6,0.61,0.62。，那么很显然就会造成有的在阴影有的不在。



\*如何解决呢：1：可以增加阴影图的分辨，2：可以给深度图中的深度加个bias，0.005-0.05就够用了。

**如何产生并解决Peter Panning：**

\*因为进行了偏移后，会造成悬浮问题。在绘制阴影深度图的时候把背面剔除改为正面剔除。

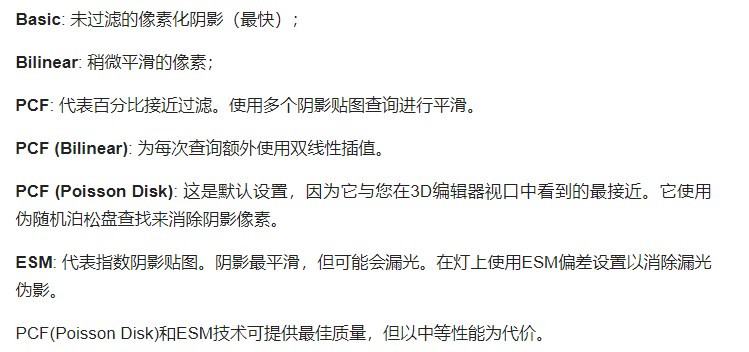
**深度图的格式如何选择呢：**

\*要选择32位数来存储，因为是float量。

**PCF是什么，用来做什么：**

\*起初用来做抗锯齿，一般常用语shadowmap消除阴影锯齿，或pcss实现的核心思想。Shadowmap是对阴影图采样，然后比较深度。Pcf是是对图周围的n\*n范围进行采样并比较深度，然后对分别采样比较的结果进行平均或加权平均。

\*双线性插值pcf：就是在采样时拿到shadowmap它上下左右四个像素点去做采样并比较深度，然后做双线性插值。最后在做pcf，这样pcf的filtersize可以小一点，但采样数未必会少。



**PCSS是什么，实现细节：**

一 计算并存储采样点被blocker挡住的深度平均

二 通过被（一）中存储的平均blocker深度去决定用多大的fileter

三 去做pcf

**ESM是什么，实现细节：**

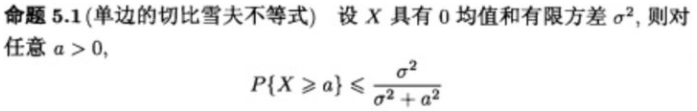
因为使用step无法实现软阴影，非0即1，ESM是用指数函数方法去防止01突变，去拟合step达到软阴影的效果哦。

Shadowmap中存 exp(C\*depth)，接收阴影时计算exp(-C\*d)，阴影结果为exp(C\*depth)\*exp(-C\*d)。

当然，当C小的时候会有严重的漏光，因为拟合step函数差远了，当C很大时可以改善漏光，但同时会导致阴影变硬（需要使用prefilter操作来让他变软），同时C很大会导致纹理通道的精度不够用。

**VSM是什么，实现细节：**

也可以实现软阴影，使用切比雪夫不等式把01之间的问题转换为概率思想去处理，有方差公式：.

课本中的切比雪夫不等式公式： 

论文中的切比雪夫不等式：，其中t = u+a,式子中要使用方差，上面又有了方差公式，所以深度图中我要存两个通道，一个存正常的深度u，另一个存深度的平方，u^2。

Vsm中也有漏光问题，因为在半影区域方差比较小阴影接近0，但多个caster的边界处由于深度变化比较大，方差增大，导致式子更接近1 从而变得更亮，出现了错误的结果。所以有一种取巧的方式，把pmax做截断，小于一个min值就截断为0，大于max截断为1。方差的最小值很大程度决定了shadow颜色的深浅。因为其特性，不太好解决漏光问题，比较适合俯视角的阴影。

Vsm漏光问题的本质是：因为公式，当方差增加时（就是两物体深度变化相差大时），就会导致上式结果趋近于1，那么就边界处就会有漏光。

同时vsm要求相机内接收所有的物体参与阴影计算，因为方差计算需要。同时要使用双通道来存浮点数，需要16位以上深度。移动端难搞（为什么难搞？？？）

**EVSM是什么，实现细节：**

EVSM同时结合了VSM和ESM，效果非常不错，但是！缺点是对rt的要求需要四通道32位深度，太耗了。

生成深度图时要四个通道分别是x=exp(constX\*d) y=x^2 z=-exp(-constY\*d) w=z^2

接收阴影时候通过一个wrap函数，wrap函数就是上面的x和z，接收时候把深度wrap为depth.x和y的两个东东，然后分别做切比雪夫min(Chebyshev(x,y,depth.x)，Cheybeshev(z,w,depth.y))，结果取最小的。

**CSM篇**

**为什么要有CSM，什么是CSM？**

已知shadowmap是在光源处搞一个相机，然后看向场景。那么如果大场景呢，（1）一个shadowmap太大的话分辨率也不够用，（2）就算够用那么光源转动时很大可能拍不全大场景。这两个问题的解决就引入了CSM。

CSM让光源相机使用跟随主相机运动，是光源相机的正交视口始终能把在主相机的视椎包含住，就可以让平行光阴影区跟着的主相机的视椎动态变化，保证阴影图的利用度最大。

流程：

\*根据cascade的个数，把主相机分成几段，求出每一段中近裁剪面和远裁剪面共八个点，把这些点变换到主相机的世界空间中去。

\*把这每组的八个点转换到灯光的视图空间下，在灯光视图空间下求出来每组视椎的包围盒。

\*然后根据近裁剪面得到的包围盒下的近裁剪面，算出包围盒的近裁剪面的中心，在把这个中心变换回世界空间，然后让相机在这个位置拍摄一次场景，为cascade1（方向和光方向统一哦）。以此类推拍n次。

\*得到n张不同级联的图，根据物体当前位置的，求出一个权重，去做计算。

**CSM效果优化与坑：**

\*包围盒最好使用正方体的：也就是灯光相机要正方形视口，因为相机在转动时，要保证两帧像素重合才能消除闪烁，如果不是正方形，投影位置改变所有像素变形会发生闪烁。

\*对相机各个拍摄点的控制：可以很好的消除抖动，根据视椎体的长和宽和shadowmap的贴图分辨率求出一个系数，保证相机的位移和视椎体的长款都是这个系数的倍数。相当于把阴影相机在视椎体的xy方向上离散化成了正方形小个子，防止相机运动或浮点数精度带来的小变换造成抖动。

\*不同级联交接处的边界的过度：不晓得00？

\*caster不在包围盒内，拍不到深度不能产生阴影，要想办法魔改。（emm就是计算出caster的包围盒，然后调整光源相机的z偏移，使得能够拍到其从而产生阴影）

\*包围盒也可以用球型包围盒，球型能改善边缘闪烁问题，具体的我还要再看一下。

\*srp调整相机参数后，使用drawrender根据cullingresult去绘制，导致调整参数后的cullingresult和之前不一致，有的caster就拍不到，这时候要重新设置cullingplane才可以。

\*要随着caster包围盒的大小去调整灯光相机的位置，那么当能拍到caster且特别远的时候，这时候精度会变得很低，当能拍到且很近的时候，这时候精度还能ok，那么这时候就会造成明显闪烁。

Unity中的半透明阴影

Unity中半透明阴影使用Dither+clip的方式去做，使用Unity自带的\_DitherMaskLOD图去把透明物体的深度图挖洞，然后采样时候再加上抖动等方式。

**深度篇：**

**什么是反向深度：**

\*因为浮点数在0附近精度最高，越往后精度越低。正常的深度图在近裁剪面部分太长，所以要搞个反向的，因为大部分物体在远处稍稍多一些，让近裁剪面部分映射的短一些，提高远处深度的精度。

**深度什么时候是线性，什么时候非线性？Unity的Linear01Depth和LinearEyeDepth如何工作的？**

**后处理篇：**

**如何实现SSAO：**

大致分为三个部分：生成半球面与随机点计算每个像素ao（计算ao），模糊/滤波，结果混合

具体细节大致为：

\*计算ao：

需要使用\_CameraDepthNormalsTexture来获取观察空间的深度和法线，然后生成法线方向的半球体与球体内一些随机采样点，遍历这些所有采样点将其变换会屏幕空间，再通过\_CameraDepthNormalsTexture获取对应观察空间的深度，进行比较。期间可以规定范围用smoothStep或Lerp进行采样效果的平滑，平滑参数为采样分为比采样点和原像素点深度的差值，最后把结果进行平均，结果为该像素点的ao值。

\*模糊滤波：

\*结果混合：

**管线篇：**

**剔除和裁剪了解什么？：**

\*先后顺序分别为：视椎体裁剪，视口剔除，面剔除。前两个都在齐次裁剪空间进行，也就是mvp之后，透视除法之前。最后一个在ndc可以

视椎体裁剪可以推倒出视椎体的矩阵六个面，判断世界空间的三角形是否在六个面之内。这是cpu阶段剔除

视口剔除主要用到Sutherland–Hodgman多边形裁剪算法，这期间会形成新的三角形，一定不能在ndc进行。因为中所周知透视投影之后w会变成-z，如果刚好有三角形超过近裁剪面之后，那么这部分的z是正的（本应该是负的），透视除法之后就会把w变成负数，这时候xy的结果就会发过来，会导致错误的显示在视椎体内。

面剔除在ndc进行。Opengl逆时针方向为正面，顺时针为背面。这时候观察方向是(0,0,1)，选择三角形任一点，逆时针方向求得到其他两点的向量，两向量叉乘，结果如果和(0,0,1)点乘小于0剔除，否则不剔除。

**说说常见的剔除操作有哪些**

图元方面的：视椎体剔除，视口裁剪，背面剔除，深度测试也算剔除吧

对象阶段的：遮挡剔除，hiz剔除

**Early-Z是什么，什么情况会失效？Pre-Z呢：**

\*这是一种优化方式，原本深度测试在片段着色器完成之后进行，现在提前到PS之前，减少PS阶段的计算量，由硬件自动进行。开启alphatest、透明度混合、关闭深度测试（early-z是建立在深度测试开启的基础上进行的）、开启Multi-Sampling（采样会印象周边像素，提前无法知道是否被剔除）、discard像素、在PS修改深度、其他任何操作要混合后面的颜色都会导致Early-Z失效，因为要保证之后的深度不变才能保证结果的正确性，否则硬件会关闭。

\*Pre-Z是软件层的方法，用来优化earlyz上述不稳定的情况，通过第一个pass实现最简单的shader，只写入深度，不写入颜色不进行复杂计算，第二个pass关闭深度写入开启深度测试。

\*可用pre-z优化草渲染，在prezpass开启alphatest，只通过opaque的片元并写入深度关闭深度测试，非opaque的东西会通过alphatest被干掉，最后只通过opaque的深度。当等到第二个pass时GPU会再次开启earlyz，只开启深度测试关闭深度写入，并设置测试通过模式为equal，这样就只保证最终只有不透明动动走进ps阶段去计算。

那么同时prez的缺点就是不太适用顶点超级多的物体，最适合就是那种顶点少，但ps阶段要复杂计算的物体。如果vs有复杂计算例如顶点动画等，同时prez会造成DC翻倍，因为是个双pass的方法，所以不能进行动态和批，如果这种情况下，反而可能会造成负优化

**重心坐标系做光栅化的优点是什么？如何进行其他顶点属性的插值？**

\*优点：适合现代GPU的并行方式，对三角形的光栅化十分快，求出的三个值可以用来其他顶点属性的插值。使用现代光栅化2x2的结构。同时2x2光栅化的优点有：\*0\*简化和加速像素分派的工作。\*1\*精简SM的架构，减少硬件单元数量和尺寸。\*2\*降低功耗，提高效能比。\*3\*无效像素虽然不会被存储结果，但可辅助有效像素求导函数。

\*怎么做：判断点和三角形的关系使用叉乘就可以，一个点和一个边的关系扩展到一个点和三个边关系。

\*其他顶点属性怎么插值：其他顶点属性插值用返回的三个重心值分别乘对应三个顶点的属性就可以了。其中这些属性都要进行都不是经过project变换都不是线性的了，要进行透视插值矫正和恢复才能得到正确的值，这些属性和1/z相关，具体推倒之后在写吧。

**半透明渲染篇：**

常规半透明渲染怎么做？优缺点？

常规的使用多pass方法做，因为半透明渲染需要关闭深度写入但要打开深度测试是，因为透明物体时在opaque物体之后渲染的，打开深度写入会导致错误的结果，所以需要通过深度测试+alphaBlend来做半透明渲染，如果像草、头发这种自身不会出现半透明可以用alphaclip去做，但效果超级差，像是被狗啃了。

一般用多个pass，方法有下面几种：

一：

pass1打开深度测试与深度写入，使用clip剔除掉透明部分，保证半透明的实体部分可以正确的渲染，但不输出颜色（ColorMask 0）。

Pass2打开深度测试关闭深度写入，主要处理半透明部分，然后做正常渲染

二：

Pass1打开深度测试与深度写入，使用clip剔除掉透明部分，保证半透明的实体部分可以正确的渲染，但不输出颜色（ColorMask 0）。

Pass2和Pass3和上面的pass2一样正常渲染，但是分别来处理背面和正面（特别是头发的渲染，有双面不同的表现，同时因为半透明的性质，分别来渲染背面和正面的结果是比较正确的）

三：（优化方案prez）

Pass1使用prez来优化（但具体要不要优化还要分析性能瓶颈再说），打开深度测试和深度写入，后面的pass2和上面的pass1一样，但不写入深度，只做深度测试同时深度测试比较方式为equal。后面的pass3 pass4和上面的pass2pass3一样了。

depthPeeling怎么做？优缺点？

优点是效果稳定不出错，缺点是很慢，很费显存，需要mrt，需要多个pass且每个pass都要做frag的效果，要剥离多少层这个是无法确定的。

做法：对一个物体渲染n次，每次都要把这次渲染结果的颜色和深度返回（要用到mrt技术），下次渲染使用上次渲染的深度图，和这次的深度做比较，如果大于（根据平台决定）上次深度，说明是下一层则渲染，把结果返回并丢弃掉这层。如此循环。

最后再分别把每层（从最后向最前混合）结果和屏幕结果原图混合。

**Forward+管线渲染篇：**

**Forward+的实现细节是什么？**

对forward的改进，利用了GPU的并行计算能力去剔除光源，将最终对像素有影响的光源列表传递给最终渲染着色器。

Forward+在传统的forward方法在最终的着色器阶段前添加添加了个光源剔除阶段（light-cullingstage）。Forward+包含三个阶段：depth prepass，灯光剔除，最终着色。Forward+对存储灯光信息的数据结构也做了修改，将光源信息存储到一个线性布局的缓存中，方便进行光源剔除和在最终的着色阶段进行访问。ZprePass是forward的可优化项，在forward+中可以大大减少最终着色器的压力。如果将zprepass和延迟渲染的Gprepass相比，只要计算深度，代价也更小。

**Forward管线渲染篇：**

前向渲染的实现细节是什么？不知道，但是主要打算说一下Unity里的forward

Unity里forward渲染时因为每个物体的着色都会考虑所有的灯光（即使没有贡献光照），所以forward中，对最最亮的灯处理为逐像素处理（可设置），对之后最多四个灯做逐顶点光照，然后剩下的做球谐光照（一种近似模拟的光照）。

**Deferred渲染篇：**

**Deferred的实现细节是什么？**

传统的是所有流程走到ps阶段，然后进行计算，在ps之后才做深度测试。Deferred的思路就是分两个pass，第一个pass把灯光计算所需计算的东西存到GBuffer中，例如（depth，normal，position，diffuseColor等），这些图是利用MRT写入Gbuffer的，同时由于光栅化之后就可以得到深度值了，这时候做深度测试把看不见的片元剔除掉。第二个pass为lightpass，在lightpass中使用存储下的结果做光照计算。

\*MRT有个非常严格的限制就是所使用的Texture必须要有相同的bit-depth

**Lightpass要做什么？**

**Unity的延迟渲染：**

msaa和ortho投影不能使用

**延迟渲染篇：**

**延迟渲染和前向渲染的区别：**

\*延迟渲染大致流程：

有两个pass来实现：第一个为geometryPass，先进行一次场景渲染，获取各种对象的几何信息存储在GBuffer中，存储的例如有Position、Color、Normal或等等其他。第二个为LightingPass，使用GBuffer纹理中的数据，在这个pass进行光照的计算。十分适合处理多光源。

\*延迟渲染的问题：

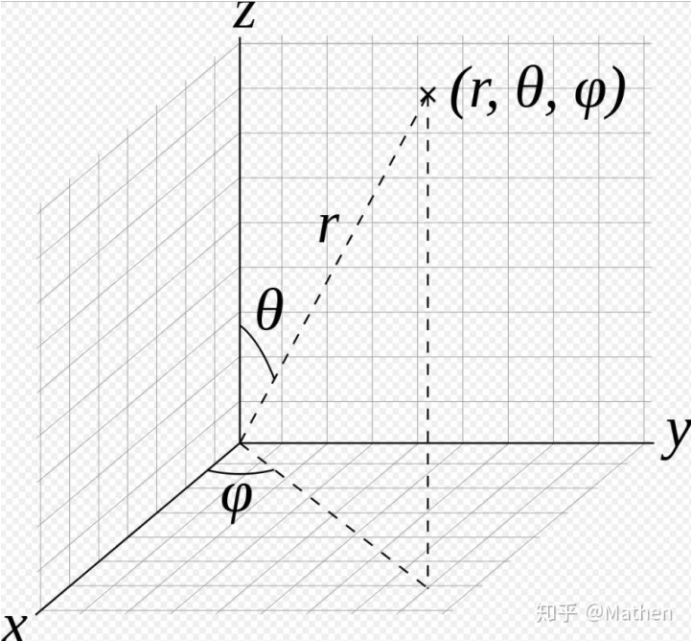
GBuffer在进行复杂场景渲染时要存储更多的东西，会消耗很多的显存

因为GBuffer太大了，做msaa需要四倍的显存，太大了难以实现

只能对所有模型进行统一的光照着色，使用一套光照算法

**数学相关知识篇：**

球面坐标：



天顶角：半径R和z轴的夹角

方位角：半径的投影R\*Sin(

**美术相关知识：**

**饱和度：**

\*指颜色的纯度或强度，简单来说就是颜色中灰色含量的高低

\*高饱和度高亮度的颜色适合做引导性的按钮

\*低饱和度低亮度的颜色适合做长时间观看的颜色

**对比度：**

\*指颜色最高亮度和最低亮度的比值，比值越大颜色明暗差异越明显

\*对比度越大让图像越醒目，反之颜色会灰蒙蒙的

**锐化：**

\*突出图像边缘的部分，让图像棱角变得清晰

**GPU方面：**

**为什么移动端选择TBDR？**

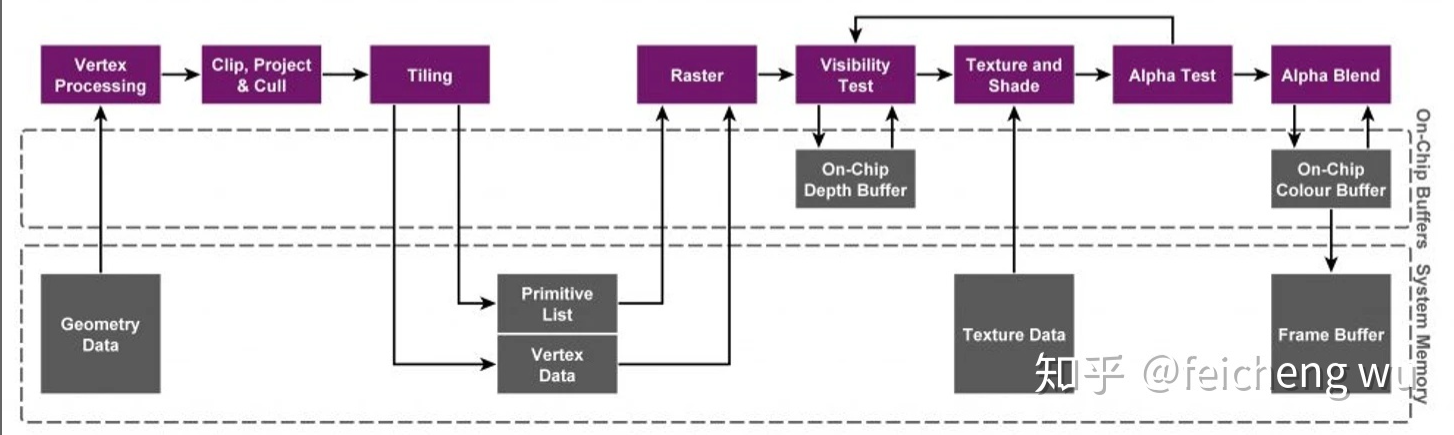
参考网址：<https://zhuanlan.zhihu.com/p/112120206>

TBR\TBDR（Tile-Based(Defferred) Rendering）主流移动端GPU架构。PC上为IMR(Immediate Mode Renderers)。和软件层面的延迟渲染不同，这是硬件层面的延迟渲染，缓存所有绘制指令最后才开始执行。

因为移动端的一大瓶颈是功耗(发热量,耗电量)，对功耗印象最大的是带宽！渲染一帧图像，对framebuffer的访问是惊人的(个中test，blend，Msaa，overdraw)等。通常gpu的onChip Memory很小（也是SRAM或L1,L2 cache），这么大的framebuffer要存在里gpu较远的DRAM(显存)上，可以把gpu想象成你家，SRAM想象成小区便利店，DRAM想象成市中心超市，从gpu对framebuffer的访问就相当于一辆货车大量的在你家和市中心之间往返运输，带宽和发热量之巨大是手机上无法接受的。

所以移动端GPU把巨大的framebuffer分成许多个小tile，是的每个小tile可以被最近的SRAM容纳，tile有多少取决于你硬件的SRAM的大小。这样分块把整个framebuffer处理完后，再整体转移到DRAM。而电脑上直接对DRAM的读写速度是非常快的，不用TBR。

TBR的来源上面说了，那么TBDR中的D是怎么加入的呢？因为改为TBR之后，移动端GPU接收CPU绘制指令的行为完全改变了。移动端是不能来一个commandbuffer就执行一个，任何一个commandbuffer会影响到整个framebuffer，那么GPU可能会在每个一个drawcall上都来回搬迁所有的tile，太慢了！TBDR是指对CPU过来的commandbuffer做vertex处理，然后对vs的结果做暂时保存，等到非要刷新整个framebuffer的时候，才做光栅化。什么是非要整个刷新framebuffer？就是GPU觉得不得不把这块framebuffer绘制好的时候。Vertex和geometry之后的数据就存放在frameData中存下来排好队。有了这个东西，还可以使用HSR硬件进行优化，专门对这些framedata做处理，找到这次渲染真正有可能会被写入到Framebuffer上的那些drawcall，而过滤掉大部分的drawcall。例如对于不透明物体，一些可能不通过ztest的，一些会被stencil reject的，



PowerVR的GPU架构是TBDR，其他都是TBR的，因为产权问题，其他TBR也有了第二个D，但是不能这么叫，所以基本行目前市面上手机都会TBDR了。，但其实这两个都是种硬件的延迟渲染的架构。

第一个D是所有TBR/TBDR必须要有的，就是分块过程binning，决定每个图元由哪些个tile去进行渲染。

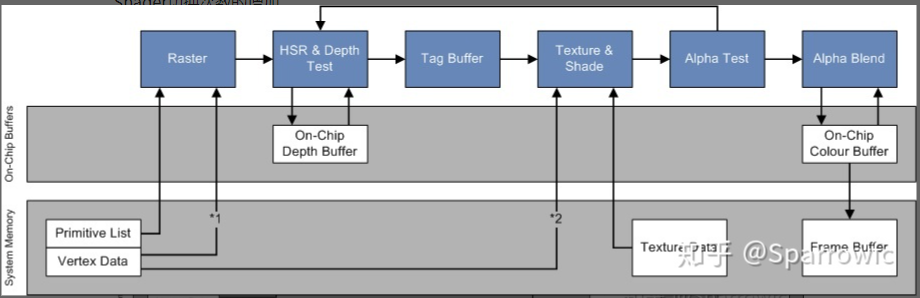
为什么只有PowerVR叫TBDR呢？因为TBDR在Raster之后还加了一个延迟。这个延迟叫HSR（Hidden Surface Removal），所以TBDR的这个D就是指的这里！！一定要搞清楚。

\*TBR：VS - Defer - raster - PS

\*TBDR：VS - Defer - raster - Defer – PS

**什么是HSR？做了什么？**

上面说了HSR就是TBDR中在raster后新加的一个延迟D。那么这个D做了什么呢？



HSR主要可以解决些overdraw问题，因为传统的early-z虽然可以优化性能，但完全不能处理overdraw问题,HSR可以看做是一种改进版的earlyz，各大厂商叫法也不同。PowerVR：HSR，Adreno：EarlyZRejection，Mali：FPK。这是因为我们在真正对一个复杂场景去渲染的时候是不可能进行严格的由近到远的绘制的。一个面积很大的地块与远处的物体相比谁应该算“近”呢？ 一个有凹面的物体，其上每个图元的的渲染顺序也会因视角不同而会出现先绘制远处三角面再绘制近处的情况。想把物体拆碎？那意味着drawcall增加。完全严格的前后物体排序？那很可能意味着Shader切换次数的增加

这里HSR登场了，不需要在软件层面对物体进行排序，HSR在硬件上实现了零Overdraw的优化。原理也超简单，当一个像素通过了EarlyZ准备执行PS进行绘制前，先不画，只记录标记这个像素归哪个图元来画。等到这个Tile上所有的图元都处理完了，最后再真正的开始绘制每个图元中被标记上能绘制的像素点。这样每个像素上实际只执行了最后通过EarlyZ的那个PS，而且由于TBR的机制，Tile块中所有图元的相关信息都在片上，可以极小代价去获得。最终零Overdraw，毫无浪费，起飞。

如果在HSR处理不透明物体的过程中突然来了一个AlphaTest的图元，那么为了保证渲染结果正确，HSR就必须要终止当前的Defer，先把已标记好的像素都绘制出来，再进行后面的绘制。这显然严重影响了渲染的效率，也是为什么官方文档特意提到尽量避免AlphaTest的原因。相对应的AlphaBlend同样也要中断HSR的Defer，强制开始绘制，但是比AlphaTest好那么一点点的是他不影响后续图元并行地继续开始进行HSR处理。这里仅是和opaque的物体比效率会下降，但和他其他GPU架构处理透明物体不见得会下降。

**GPU是如何与CPU协调工作的？**

**GPU也有缓存机制吗？有几层？它们的速度差异多少？**

**GPU的渲染流程有哪些阶段？它们的功能分别是什么？**

**Early-Z技术是什么？发生在哪个阶段？这个阶段还会发生什么？会产生什么问题？如何解决？**

\*可以在fragment之前拒绝无效像素进入，early-z剔除的最小单位不是一个像素，是2\*2的像素块。early-z完成后，最后还会进行深度测试。

**SIMD和SIMT是什么？它们的好处是什么？co-issue呢？**

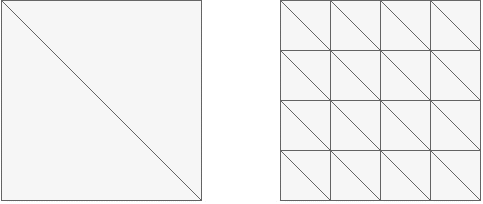
**GPU是并行处理的么？若是，硬件层是如何设计和实现的？**

**GPC、TPC、SM是什么？Warp又是什么？它们和Core、Thread之间的关系如何？**

**顶点着色器（VS）和像素着色器（PS）可以是同一处理单元吗？为什么？**

**像素着色器（PS）的最小处理单位是1像素吗？为什么？会带来什么影响？**

**Shader中的if、for等语句会降低渲染效率吗？为什么？**

**如下图，渲染相同面积的图形，三角形数量少（左）的还是数量多（右）的效率更快？为什么？**

**GPU Context是什么？有什么作用？**

**造成渲染瓶颈的问题很可能有哪些？该如何避免或优化它们？**

算法方面：

**屏幕空间深度重建世界坐标：**

\*取出来的depth，如果是opengl平台，要记得depth\*2-1（因为opengl深度范围在[-1,1]），dx的就不用了，因为dx本来就在[0,1]

**杂项：**

**哈达玛积：**

\*就是颜色乘法用的

**延迟渲染：**

\*延迟渲染中Gbuffer参数

CameraDepthTexture 纹理采样 深度缓存 Depth

\_CameraGBufferTexture0 纹理采样 漫反射 Diffuse

\_CameraGBufferTexture1 纹理采样 镜面反射 Specular

\_CameraGBufferTexture2 纹理采样 世界空间法线 World Space Normal

\_CameraGBufferTexture3 纹理采样 (HDR)格式 Emission + lighting + lightmaps + reflection probes buffer

CameraGBufferTexture4 纹理采样 存储了shadowmask信息

\_CameraDepthTexture

采样出的结果经过MVP且经过透视除法后的值

\_CameraDepthNormalsTexture

需要用DecodeDepthNormal来解码法线和深度，这里的法线是相机空间法线[-1,1]的。深度经过01映射后的01深度。

**局部光照模型：**

\*lambert漫反射模型：

环境光部分： I\_ambient = Kd\*Ia (kd(0-1)材质对光的反射系数(也就是直接采样纹理颜色)，Ia为环境光颜色)

直接光部分：I\_dir = Kd\*max(0,dot(N,L))\*I\_i\*I\_c (I\_i为光强度，N为归一化法线，L为归一化光方向，I\_c为光颜色)

\*halfLambert漫反射模型:

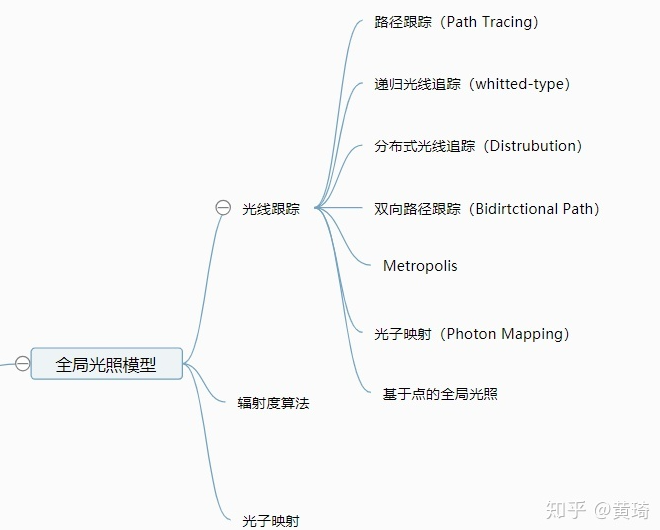
\*gourand光照模型

\*phong镜面反射光照模型

\*BlinnPhong镜面反射光照改版模型

\*CookTorrance模型

全局光照模型：



**优化篇：**

**Drawcall的压力是在CPU还是GPU？具体是什么问题导致压力？**

**如果一次上传上万个顶点，和每次上传几个但上传1000次，哪个快？**

Drawcall代价其实并不高，旧版本的dx需要切换上下文所以代价比较高，主要原因是GPU变换和绘制三角形的速度要比GPU提交的速度更快，如果每次只提交一些三角形上去，那个GPU大部分时间都是闲置的。但drawcall的主要耗费成本是每次cpu提交数据太少，CPU不能喂饱GPU。。。所以感觉Drawcall的瓶颈主要在于CPU提交的数据量上。

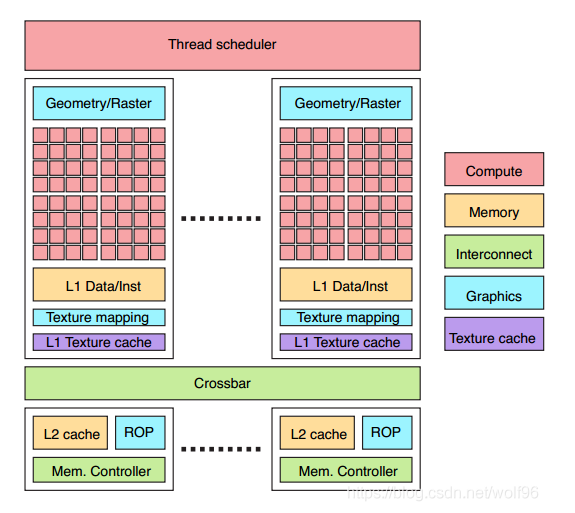
什么是SetPassCall？

是消耗的重点，一个材质球引用的shader可以有多个pass，因此携带这个材质球的物体会被渲染多次，"激活"一一个pass的过程就是set pass call，改变pass的行为都会造成setpasscall，消耗要比drawcall大的多，用来传递并改变RenderState，一般指setShader、BindProperty、BindTexture这套，就是吧当前用到的材质信息，贴图信息，shaderpass等信息传递到GPU上。而drawcall是下命令让GPU绘制三角形（drawcall其本身并不太昂贵），现在PC上，1ms处理1k-2k个drawcall也正常，多个drawcall未必比合批慢。

大幅度提高Drawcall的办法，使用Unity的DrawMeshInstancedIndirect，可以省略掉许多复杂的消息传递过程，不由CPU指定绘制目标，而是有GPU自行支配，可以把模型数据长存在显存中，比如贴图或顶点数据等，然后indirectDraw会提供给vertexshader一个索引整数值，开发者要手动在shader中读取显存数据，这样在绘制静态物体时，会有非常大的提升。

**Texture Cache**

在high level看，GPU很多sahder处理内核，笔记本或或平板可能只有几个，高端pc机有十几个。除了shader核还有纹理单元，一个shader核有一个纹理单元，但他可能被多个shader核共享，纹理单元中纹理缓存。



整个芯片共享一个L2缓存，每个单元有自己的L1缓存。当纹理单元和shader核的运算相互独立，纹理单元是异步的，当sahder读取一次纹理时，会向纹理单元发送请求去，shader可能挂起等待结果返回。纹理单元要做很多事，如（mip，各向异性，uv坐标转换为纹素坐标，应用clamp和wrap模式等），然后在对应的缓存中去找（L1中没有找L2，L2没有找DRAM）。如果没有命中缓存，需要从外部存储读取到新的缓存线中，所以会有很大延迟。

光栅化是以2x2的quad为单位的，如果三角形太小只包含一个像素，那么会画上多余的三个空像素，效率变低。

**传统的扫描线光栅化的问题与为什么选贼tile：**

1.贴图中纹素的访问顺序由光栅化顺序来决定，如果cache中存储的方向和光栅化不同，就会导致cache命中率下降，加载新的cacheline。

2.如果三角形过大，水平绘制完一行，纹理缓存被全部填充，到下一行又被清空。

使用Tile的光栅化会解决上面的两个问题。因为如果要双线性过滤和求ddx ddy的话，需要四个像素。

**Texture cache的优化**

文章：<https://blog.csdn.net/wolf96/article/details/87884209>

TextureCache是存储一个图片数据的只读cache。

现代tile的gpu对mipmap和双线性filter支持非常好，mipmap甚至还有优化的效果，可以更好帮助命中cache。

不按照shader插值的uv读取贴图会破坏纹理缓存，导致经常cache missing，进而经常刷新缓存。

贴图太大会导致cache超载，导致cache missing。贴图的长宽最好要是2的n次方。

压缩贴图可节省带宽，因为cache中存储的都是压缩的texel，能增加纹理缓存中存储的texel数量，解压缩是发生在纹理单元中，读取命中缓存纹理和纹理filter之间。

少用纹理三线性过滤和各向异性过滤

**移动端根据GPU架构去优化**

文章：<https://zhuanlan.zhihu.com/p/112120206>

了解了移动GPU的设计架构，就能明白：移动GPU相对于桌面GPU来说最大的区别就是带宽的开销成本很高。因此任何有高带宽开销成本的操作都具有高性价比优化的潜力。

降低Shader的复杂度固然是通用的降低GPU开销的方式，但如果还考虑到耗电或发热问题的话，带宽开销应该是最为重点的关注。

针对于带宽的优化具体来说可以有以下几点：

\*贴图格式能压缩就压缩。贴图内存越小，片上命中率就越高，总的传输量也少

能开mipmap就开mipmap（前提是能用到，UI贴图就不用开了）。与减小实际使用的贴图内存是一个道理，但是会增加总贴图的内存占用大小，需要在内存开销和带宽开销上做一个平衡。

\*随机纹理寻址相对于相邻纹理寻址有显著开销。提高片上命中率。

\*3DTexture Sampling有显著的开销。3DTexture整体内存占用大，垂直方向相邻像素内存不相邻很容易cache miss，这是我个人推测。

\*Trilinear/Anisotropic相对于Bilinear有显著的开销。Trilinear其实就相当于tex3D了(此结论不负责任)，Bilinear相对于Point几乎没有额外开销（此结论负责任，texture fetch都是一次拿相邻的四个出来），所以Bilinear能忍就尽量凑合用着吧。

\*使用LUT（look up texture）很可能是负优化。需要对比权衡带宽占用+texture fetch操作增加与ALU占用增加降低并行效率，另外还很可能涉及到美术工作流和最终效果，所以是个不是很好进行操作的优化。之前看过腾讯的技术分享将引擎中Tonemapping那步的3DLUT（UE4和Unity都是这样的）替换为函数拟合的优化，理论上应该是会提升不少性能，但是要想真正应用到生产环境，保证效果，还要做好拟合工具链，是得费不少力气的

通道图能合并就合并，减少Shader中贴图采样次数。这个不多说了

\*控制Framebuffer大小。这个也不多说了

\*总顶点数量也是带宽开销的影响因素。虽然以现在GPU的计算能力来说，顶点数增多产生的VS计算开销增加通常是忽略不计的。但是仍不能忽略总顶点数量对于VertexBuffer所消耗带宽的影响，对于总顶点数的限制应该更多的从带宽消耗上去进行测试和分析。

另外还有一些优化点，一般来说引擎都给弄好了，不是特别作应该都碰不到，也简单列一下：

\*任何必要情况下（比如新一帧绘制前）都应该显式地对FrameBuffer进行Clear操作。

不要在一帧的绘制中多次更新VBO。

**Shader if和for效率讲解**

参考资料：<https://blog.csdn.net/xiaoyafang123/article/details/84942201>

<https://blog.csdn.net/lou09020207yes/article/details/76033977>

<https://zhuanlan.zhihu.com/p/122467342>

\*If和for确实有效率问题，但难以避免，也就无需处理

\*用step代替if大多是无优化或负优化，少数情况有提升

\*可以用shader变体来取代一些只和固定参数有关的if。

\*if之所以慢是因为GPU的“进行计算的原作单元”和“指令执行的逻辑单元”不是一一对应的，而是一对多的关系，执行一个指令是并不是操作一个像素，而是一组像素，并一条一条指令向下执行。如果有像素有一个走到了if的true，那么其他像素没有到true，虽然这次不用计算，但没有其他的逻辑单元可以和他们配对工作，所以也只能干等。即使有个一像素需要计算，也要占用全部资源。所以if分支的效率问题是会导致多个分支重复执行。。

\*虽然用宏定义去切换变体能稍稍好一些，但是变体太多也可能会有又内存和加载时间的问题，来回切换变体也有问题，所以更具实际情况去选择。如果if中的值是uniform传来的，那么这个值在这个pass一定是不变的，这时候再用step就蠢死了。

\*if中是uniform传来的，或#if#else，或if中是const值，叫做静态分支，性能几乎无损。其他的ifelse就是动态分支。

\*GPU这种工作方式叫Single Instruction Multiple Threads(单指令多线程)，前面用一个像素说法不准确，因为顶点的处理也是一样，应该叫一个thread，被称为“线程簇”，也就是一般说的wrap。硬件一直在发展，很难判断if的成本有多少。自己测试才靠谱，没事不要瞎优化。

在shader中使用if会打断GPU的warp内部（或wavefront）的并行化。

**天空盒：**

**天空盒的绘制时机是什么时候：**

天空盒的渲染顺序属于background，要最先被最先渲染的。

**天空盒的实现细节：**

在实现软渲染器时候，一定要视椎体裁剪和视口剔除才能做天空盒啊，要不然会出现很离谱的问题。

启用正面剔除或不剔除。把o.windowpos.z = o.windowpos.w，保证透视除法后在远裁剪面上。深度测试使用小于等于，因为在远裁剪面上如果是小于的话绘制不出来。

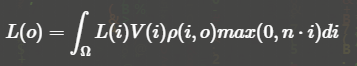
**IBL和PRT：**

**什么是IBL：**

Ibl是基于图像的照明（俗语），核心思想就是预计算，本质是split sum，主要计算的部分有：间接光漫反射，间接光镜面反射，其中又包括brdf积分表（u：（n.dot(立体角)）,v:(粗糙度)）。缺点是：难解决阴影的遮挡问题，目前工业界主流方法是为cubemap最亮区域生成一个主要阴影。

什么是PRT：

Prt稍复杂些，它把渲染方程的辐射率L和brdf项展开成球面谐波函数为基的系数权重，预计算就是如何把函数投影到球面谐波函数表达的空间。分为两部分：Li为lighting项，后面都为transport项



**数学：**

**如何判断一个东西身前还是身后：**

\*点积，我当前的forward和我到敌人的方向(tarPos-selfPos)，做点乘，大于0表示在前面，小于0在后面，等于0在左或右

怎么知道敌人在左还是在右？左手叉乘：forward X 我到敌人方向 z为正，敌人在右，反之在左

**胡扯mvp变换：（我好想悟了）**

世界空间是Unity标准左手系

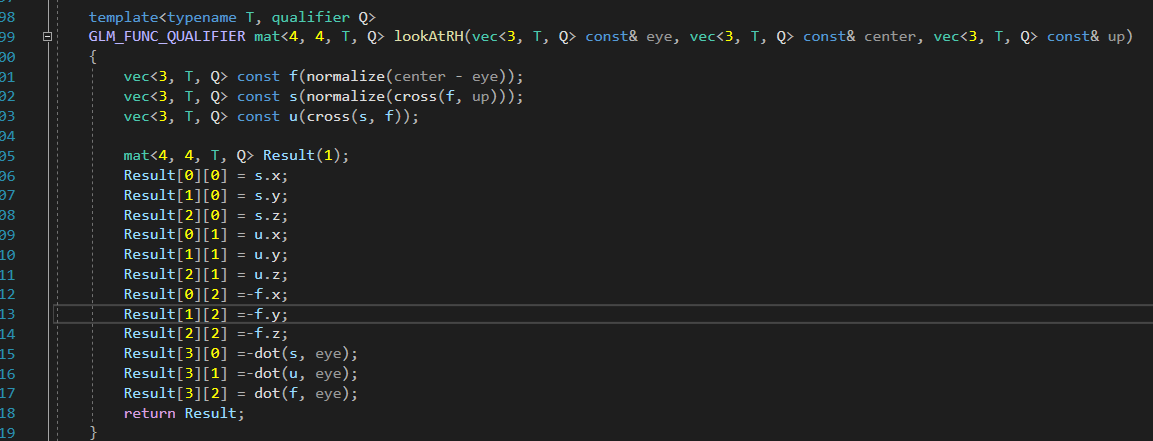
一个模型的点P为(0,0,0,1),加入M矩阵为单位矩阵，也就是不进行变换，那么直接启动后面的VP变换。

相机在(0,1,-10)，看向z的负半轴

使用lookat矩阵建立相机的基坐标：

相机的方向向量dirVector = 看向的方向反向，所以其实是看向正半轴的。

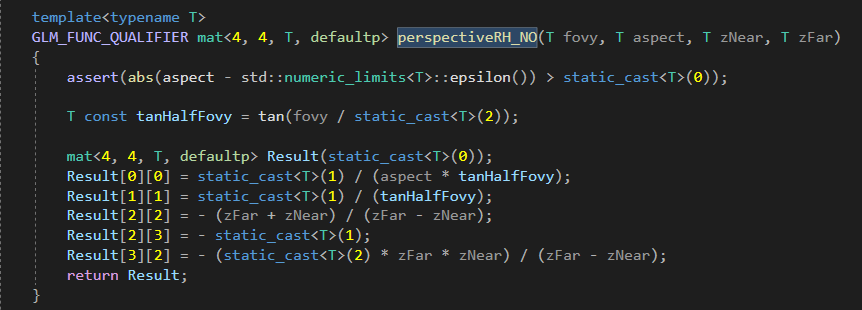
Unity用的下面这个view矩阵（view矩阵有两个版本，左手和右手，unity应该是下面这个右手版本）。。发现一通重建之后相机的基坐标是个右手系的，并不是Unity中使用的左手系，所以需要对f向量取反保证手性恢复到左手，也就是下面view矩阵所干的事。



上面发现，如果点通过M矩阵后在正半轴的话，通过V矩阵后，全都搞到负半轴去了，为什么要这么搞呢。当然有原因，应为为了下面的透视投影。

P矩阵有四个版本（ogl左手，ogl右手，dx左手，dx右手），Unity应该用的下面这个版本，我们知道通过V之后，坐标都搞到负半轴（其实都worldPos都取反了，暂时就这么说了，好理解）,透视投影又可以搞回去。透视投影发现[2][3]有个-1，当P变换之后，w=-Z(view)，

因为要进行透视除法，所以要保证这个w要为正数。透视除法其实是正真的投影，它和深度密切相关，深度越大，物体的xyz就越小，反之越大，它才可以让物体有近大远小的性质。



**ComputeShader：**

ComputeBuffer：（大小，多少位）

例如：var xx = new Data[100] {a=1,b=2,c=3} 那么，ComputeBuffer(xx.length,4\*3)，一个data有三个字段，每个字段4字节

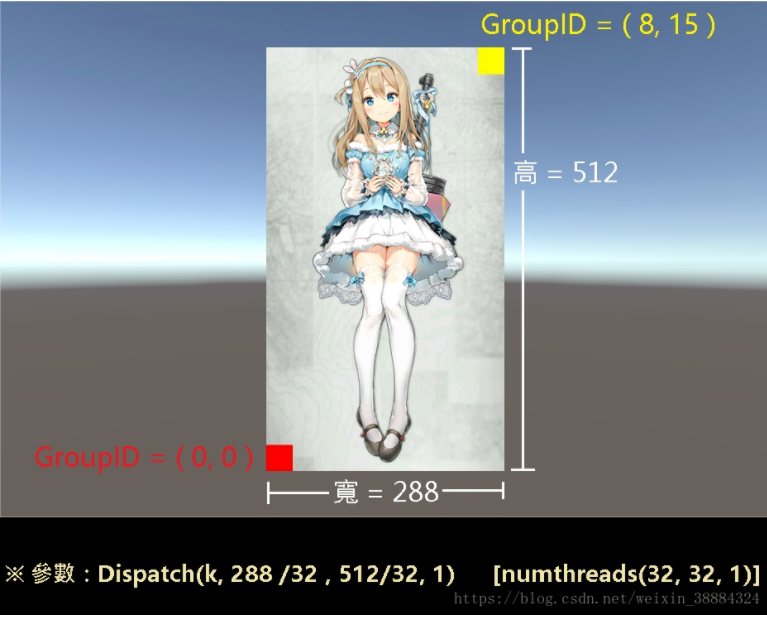
ComputeShader访问Texture的mipmap：xxx.mips[level]

Dispathc(x,y,z)，可以理解为决定使用多大范围的线程组

[numthreads(x,y,z)]如果有一个图像为256\*128，那么只要是dispatch.x\*num.x = 256 , dispatch.y\*num.y = 128的，都能让图像完全使用起来。但是num.x\*num.y\*num.z必须小于等于1024.



SV\_GroupID：线程组ID，如下图，是最大的部分，一块一块的，dispatch决定了使用多大范围的线程组



SV\_GroupThreadID：对应到每个线程组内每个线程的子ID，（三维可理解为group组内坐标），如下所示，每一个组内的线程id



SV\_GroupIndex：每个组内线程id的索引（一维），如下所示，下图每个线程组有32\*32个线程，一次排列的就是他们的索引



SV\_DispatchThreadID：唯一id，可以理解为整张图的坐标

DispatchThreadID = groupID\*numthreads + GroupThreadID

**什么是TBDR：**

好文章：<https://blog.csdn.net/leonwei/article/details/79298381>

是现代移动端GPU架构，传统的是IR架构（Immediately-Rendering）。改良过得是TBR（tile-based-rendering），把巨大的framebuffer分成多个小块，每个小块可以让离GPU最近的SRAM容纳，这样GPU可以分批的一块块的在SRAM上访问framebuffer，一整块访问好了之后然后整体转回DRAM上。

TBDR会在所有drawcall到了之后做一次HSR，Unity会根据芯片类型确定GPU架构看是否能做HSR。

苹果使用TBDR，安卓是用TBR，PC使用IMR（immediately Mode Rendering）

**SSAO篇**

**SSAO是什么？解决什么问题？**

是屏幕环境光遮蔽，通过场景中的几何信息来计算遮蔽关系，模拟全局光照的细节阴影，让场景更有层级感。优点是属于后处理技术，与场景复杂度无关，实现比较简单

**如何实现？**

先生成半球面的一堆采样点，然后获取到屏幕坐标的法线和深度，遍历所有采样点，和屏幕法线点乘，看是否在半球上，如果不在则翻转过来让它在。然后获取到屏幕空间重建相机空间坐标后，把每个采样点应用上去，在变换到屏幕空间，去采样获得采样点的屏幕空间深度，和原始屏幕空间深度作比较，计算遮挡关系

**缺点是什么？**

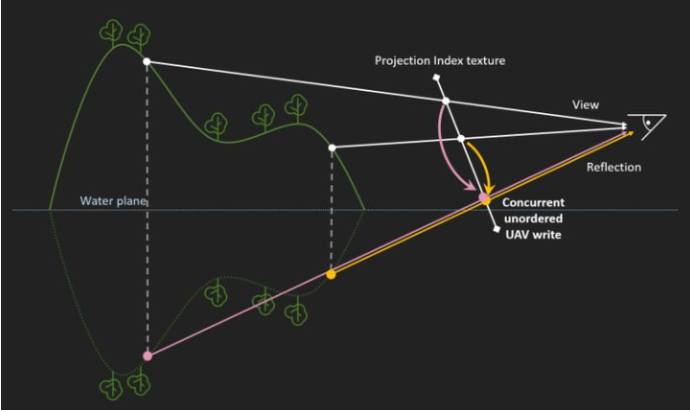
处理后需要和模糊搭配使用，效率不太好，要用很多采样点才能得到不错的效果。

**SSPR篇**

**SSPR是什么，如何实现**

\*屏幕空间屏幕反射。大致为一下这些步骤：1.获取屏幕空间深度重建世界坐标 2.获取反射平面，根据反射平面去得到翻转后的世界空间坐标 3.根据翻转后的，求出新的屏幕空间uv，然后把原屏幕的颜色写入uav。

**SSPR过程中有什么问题，如何去处理**



\*遮挡关系不正确，因为上图，俩像素点翻转后可能重合，所以为了解决重合要加个kernel去做深度测试保留正确的结果。

\*有小黑洞，上面因为可能俩像素会重合，那么原本到uav上写俩点，现在写了一个点，另一个点就空了，所以出现了空洞。为了解决这个问题，可以周围像素采样做模糊，可以判断是否空洞（当前灰度为0或周围像素和为0）然后用最近的像素去填充，

\*不在屏幕内的东西不能反射，同时也是所有ss效果的通病。（还有最后混合的时候保证效果对和判断下深度图ok不）

**抗锯齿篇**

**MSAA和SSAA分别做了什么？有什么区别？**

\*SSAA是用一张更大分别率的图，将每个像素分成四个子采样点，那么这时候更大分辨率的颜色缓存就是正常的4倍。让每个子采样点都走过ps，然后把最终每个像素内四个子采样点计算出的结果，做一个加权平均，把结果输出。

\*MSAA则是SSAA的加强版，虽同样是使用更大分辨率的图，但他却可以保持每个采样点值走一遍ps而不是四遍。具体流程是：

--很巧妙的在光栅化记录每个sample的覆盖关系，也就是光栅化三角形时候，对每个像素内的四个采样点判断并记录是否被三角形覆盖，覆盖的记录为1，反之为0。并记录插值求来的深度信息、模板信息，为了第三步的使用。

--对每个片元都走一遍ps，把结果存在colorbuffer中

--针对每个sample采样点进行深度测试，模板测试。如果通过深度测试和模板测试并且第一步记录的覆盖信息为1，那么就进入下面的解析阶段。

--每个像素有四个sample，如果两个sample进入到了这里，那么最后的颜色就是1/2ColorBuffer中的颜色。

总结：msaa可以改善几何体的走样，但会增大depth、stencil和colorbuffer的开销。

**为什么延迟渲染不能用MSAA**

因为延迟渲染在第一个geometryPass光栅化之后就已经可以得到深度从而进行深度测试，但是msaa要要在ps之后得到shading结果加之可见性判断再做处理，延迟渲染中在ps后每个三角形的深度法线等等几何信息已经丢失，留下的只是屏幕空间的信息。无法判断具体的覆盖信息，所以用在延迟渲染上结果是不准确的。MSAA仅能解决几何走样现象（边缘的锯齿感），不能解决着色走样显现（高光闪烁，高光抗锯齿也就是）。

**MLAA是什么？如何实现的？**

形态学抗锯齿（Monorplogical Anti-Aiasing）。

思路是检测每帧图像的边缘（通常可通过亮度，颜色、深度或法线来进行边缘检测）然后对这些边缘进行模式识别，归出ZUL三种形状，根据边缘形状对边缘进行重新矢量化，并对边缘上的像素根据覆盖面积计算混合权重，将周围颜色与其进行融合。也就是分为三个步骤：边缘检测、计算权重因子、混合周围像素。

**SMAA是什么？如何实现的？**

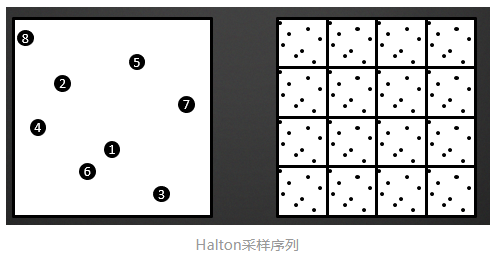
子像素形态学抗锯齿（Subpixel Monorplogical Anti-Aiasing），建立在MLAA的基础上

在MLAA的流程上改进的，有点复杂呃呃。。。

TAA是什么？如何实现？

前面说了，SSAA、MSAA通过添加采样点来实现抗锯齿，需要额外内存消耗。TAA和SSAA原理差不多，都要添加多个采样点，但是TAA通过结合历史帧，将每个像素的多次采样均摊到多个帧中，开销会小很多。可以处理着色锯齿（高光抗锯齿）

静态场景：实现方式为在每帧采样时将子采样点便宜，实现抖动（Jitter），抖动的便宜要和MSAA子采样点防止位置相同，使用底差异序列来更好的实现抗锯齿效果

，要实现对采样点偏移，需要稍微修改下透视投影矩阵，需要将便宜的xy分量分别写入到投影矩阵的[2,0]和[2,1]就可以了

**抗锯齿部分详解和实现原理：**

两篇非常好的文章，强烈建议观看：

<https://www.zhihu.com/question/58595055>

<https://zhuanlan.zhihu.com/p/415087003>

实现抗锯齿的一些思路：

第一种是提高采样频率，在每个像素中进行多次采样，根据多次采样结果来综合计算最终的颜色值，例如MSAA、TAA。

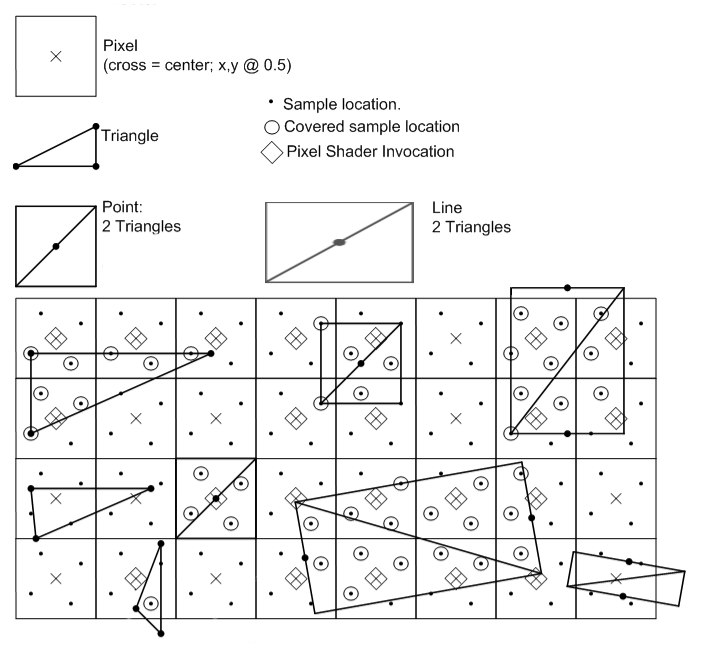
第二种是通过后处理的方式，寻找屏幕中的像素块边缘，得到边界信息将两侧像素点颜色插值得到结果颜色，例如FXAA、SMAA。

MSAA是最早出现的抗锯齿方法，主要通过硬件实现，是以前非常流行的方案。FXAA、SMAA是通过后处理方式，开销稳定，不会因为物体多少而印象开销。TAA早期用于SSR，VolumeCloud的降噪中，近些年随着延迟管线的流行，难以使用MSAA，TAA逐渐兴起。

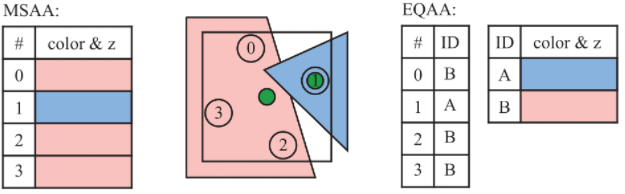
\*SSAA：想要得到抗锯齿，最简单的方法就是将图形按照xN倍的大小进行渲染，再将图像整体缩小，比如400\*600的屏幕上，先渲染到800\*1200的贴图上，然后在讲贴图缩小1/2，缩小的处理方式是进行boxfilter，提高了采样频率来达到抗锯齿的目的。这总就是超级采样抗锯齿(SSAA,super sampleing anti-aliasing)，这种的处理方式消耗非常的惊人，一般不会使用。注意子采样点不是均匀摆放的，是按照Rotated grid supersampling（RGSS）的方式来放置。

\*MSAA：同样会设置子采样点，但它就聪明很多了。对于每个子采样点，会先进行coverage test（覆盖测试），覆盖测试就是检测出像素点是否在三角形内部，如果在三角形内部说明需要采样（计算pixel shader），处于性能考虑，同一个像素上的子采样点不会都进行ps计算，具体计算方法在后面说。

当计算完成后，每个通过覆盖测试的子采样点还要进行depth-stencil test(深度-模板测试)，这个和普通的深度模板测试是一样的，只是发生在子采样点而已。当通过后，就要给子采样点写入颜色了。

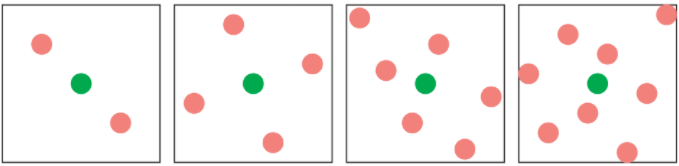


上图大概是DX介绍的MSAA的示意图，结合上图我们说一下写入子采样点颜色是怎么做的。实际采样的位置是上图的菱形处。当三角形覆盖到中心采样点时，那么被覆盖到的子采样点使用中心采样的结果。如果没覆盖到中心采样点还用中心的采样去做可能结果就是错的，那么GPU会使用centroid sampling（重心插值采样）来设置采样点颜色。**PixelFrequency**是每个pixel只执行一次，因为其他四个采样点的颜色都是通过上述规则copy过去的。当然也有SampleFrequency，每个子采样点都有对应的Ps负责。着色是1X的，但深度和模板是4X的copy颜色的时候，指挥copy给通过了模板深度的采样点，如果一个像素中所有的子采样点都没通过，那么这个像素点会被discard



MSAA用单独的贴图格式来存储，比如RGBA\_4X，表示四采样点的MSAA贴图，占用内存是普通贴图的4倍。

当上述工作完成后，最后要进行的就是resolve操作来得到最终的结果，一般来说使用boxfilter来做resolve，就是直接求个均值。同时注意，这些子采样点的位置也不是均等固定的，更多是用例如Halton、PoisonDisk等低差异采样序列去分布采样点，一般采样模式会在硬件中集成好，不要手动设置。

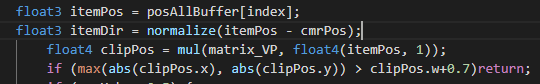


**Hierarchical ZBuffer篇：**

**什么是hiz？解决什么问题？如何实现？缺点？**

用上一帧的深度图生成mipmap，但这和常规的mipmap不同，它生成下一级别mipmap时，用2X2区域中选择最大的深度，reverse-z则取最小的。因为例如一个物体的包围盒在mip0上覆盖了16X16的范围，那么只要在mip4上比较一次，就不用比较256次了。因为如果在mip0都全部小于这个深度，那么mip4中存储的这块的最大深度，那么必然这个深度也小于mip4中的这个深度。

大佬的基于海草剔除中，用DrawInstance接口画的草，把生成的位置传到computeshader里做剔除（视椎体剔除和hiz的深度比较），然后再把剔除后的结果发回去去绘制。

Hack：视椎体剔除

缺点是由于使用上一帧数据难以处理相机快速移动的情况。CPU端回读hiz剔除结果，GPUCPU之间数据交换也是瓶颈。

杂项：

RTT（Render To Texture）渲染到纹理

通常的渲染是直接渲染到backbuffer，而rtt是渲染到纹理上，就和Unity中setrendertarget然后渲染到一张rt上结果差不多？？

SwapChain（交换链）

是一系列虚拟帧缓存区，由显卡和图形api用于稳定准数和其他作用。通常在图形内存中，也可以存在系统内存中，不使用swapChain可能会卡，基本是必须的，具有两个的是双缓存，dx中实现了三缓存。当展示当前帧的时候，dx后台会计算下一帧

功能：每个swapChain中至少有两个缓存区，第一个framebuffer，即screenbuffer（也叫frontbuffer），是呈现到视频卡输出的缓冲区。其余缓冲区称为后向缓冲区（backbuffer）。每次显示一个新帧是，交换链中的第一个backbuffer将取代frontbuffer，这称为呈现（present）或交换（swap）。可以在前面的屏幕缓冲区和其他backbuffer执行各种操作。屏幕缓冲区可能被简单的覆盖或返回交换链的后面记性进一步的处理，所采取的操作由程序决定并依赖api。Frontbuffer是只读的不可修改。Backbuffer是设备即将绘制的渲染目标，一旦完成了draw操作，swapchain就会交换两个缓冲区。

那么rendertarget是什么呢，在显示frontbuffer时，dx会在后台渲染backbuffer，这就是rendertarget。因为swapchain，front和backbuffer在一直交换，交换后back就成了front，front就成了back，rendertarget始终渲染的是backbuffer这没问题。Rendertarget可以放各种renderbuffer

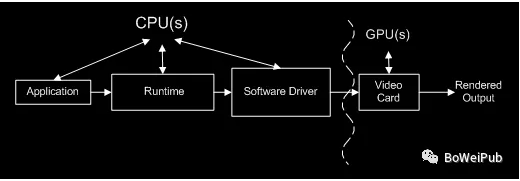
计算一张fb有多大，已知格式为RGBA24。

\*结果应该是1024\*768\*12(每个通道共24位3字节）/1024/1024 = 9M。共四个通道。

\*挺好的文章：<https://blog.csdn.net/u011476173/article/details/38712031>

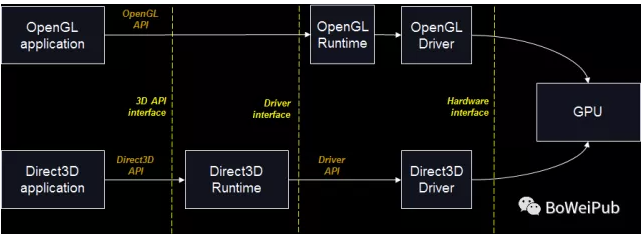
每一次调用API做了什么，CommandBuffer如何优化：

Application->Runtime->Driver->Video Card(GPU)，每一步都有一定消耗



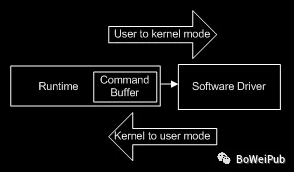
每调用一次渲染API并不是直接经过以上说的所有组件通知GPU执行我们的调用。Runtime会将所有的API调用先转换为设备无关的“命令”（之所以是设备无关的，主要是因为这样我们写的程序就可以运行在任何特性兼容的硬件上了。运行时库使不同的硬件架构相对我们变的透明。），然后将命令缓存在Runtime的Command Buffer中。

我在上一篇文章中介绍过这个Command Buffer的概念，但是当时说的是这个Command Buffer是存在于Driver中，由Driver维护的。给大家看一下Direct3D和OpenGL的Runtime的差异，相信大家马上就会明白为什么对于这个Command Buffer的位置怎么说的都有：



Direct3D的Runtime是独立的一堆DLL，我们在开发Direct3D应用的时候总是要引入这些动态链接库。而OpenGL的Runtime是和Driver整合在一起的。所以有的时候我们会把OpenGL的Runtime和Driver统称为Driver。那么对于OpenGL，我们既可以说Command Buffer位于Runtime中，也可以说其位于Driver中了。

命令从Runtime到Driver的过程中，CPU要发生从用户模式到内核模式的切换。模式切换对于CPU来说是一件非常耗时的工作，所以如果所有的API调用Runtime都直接发送渲染命令给Driver，那就会导致每次API调用都发生CPU模式切换，这个性能消耗是非常大的。Runtime中的Command Buffer可以将一些没有必要马上发送给Driver的命令缓冲起来，在适当的时机一起发送给Driver，进而在Video Card执行。以这样的方式来寻求最少的CPU模式切换，提升效率。



补充：Shader是在用户模式下编译的，并且编译成设备无关的中间码。**Driver**解析中间码，然后使用GPU硬件实现我们设计的功能。

前面的文章介绍过，渲染状态是一组全局的指导GPU工作的变量集合。如果渲染状态要发生改变，那么要使用之前渲染状态进行渲染的所有Draw call命令都必须已经被执行了。也就是说之前缓冲在Runtime的Command Buffer中的所有Draw call命令必须被刷新。这会强制发生一次CPU从用户模式到内核模式的切换。

综合以上，Draw call之间切换Texture、Shader或者Material参数会导致CPU至少两方面的时间消耗：

\*把Draw call及渲染状态切换的API调用转换成设备无关命令耗费的时间（其中还包括命令检查等操作）。

\*刷新Command Buffer导致CPU由用户模式切换到内核模式带来的时间消耗。

第一点的时间消耗其实并不大。关键的时间消耗在于第二点。所以一般情况下我们希望Command Buffer缓存尽可能多的命令，然后一次全部提交给GPU执行。

**循环展开优化 unroll：**

是什么：通过增加每次迭代计算的元素数量，来减少循环迭代次数。针对加减乘等简单运算做改进。用于迭代次数确定的循环。

优化点：\*减少分支预测失败（GPU有分支预测，但相比CPU差很多） \*减少不直接不直接有助于程序结果的操作的数量，例如循环索引计算和条件分支。

**一些Opengl中的概念：**

Vertex{

Float3 pos;

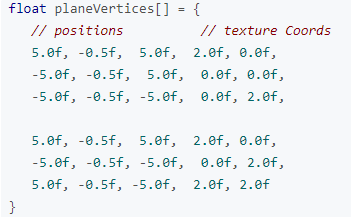
Float2 uv;

----等等

}

VBO（VertexBufferObject）：vertex都在内存中，要交给GPU去渲染需要提交顶点，VBO可以一次提交大量顶点去GPU中。

VAO（VertexArrayObject）：如果只把VBO传到GPU中，缺不知道那部分位置代表哪些顶点属性也是不行的，所以需要VAO来做这个事情。

前三个绑定顶点位置，后两个绑定顶点uv，以此类推。

**一些内置优化以及原理**

SRP Batch篇：

优化了什么问题：

**Static batching（静态合批）**

使用条件：

模型要静态，不能移动旋转缩放。勾选static的勾。

模型使用相同shader的材质

需要额外内存和包体占用

合批后mesh总索引数不能超过各自平台上线，一般为32k-64k

原理：

在Build时候Unity会自动提取这些共享材质的静态模型的VertexBuffer和IndexBuffer，根据摆放位置等信息，将这些顶点数据变换到世界空间，存储在打的VBO和IBO中，并记录每个子模型的IBO在构建大的IBO中的其实结束位置。 在后续的绘制中，一次性提交整个合并模型的顶点数据，根据引擎的场景管理判断各自每个子模型（SubMesh）的可见性。然后设置一次渲染状态，调用多次Drawcall去绘制每一个子模型。

Static batching不会降低drawcall，但可以降低设置渲染状态（setpasscall）增大提交顶点数

\*\*勾选Static的GameObject下的Mesh都会被合入CombineMesh（无论什么材质），且每个Mesh都作为SubMesh存在。在Unity 5.3之前，对于渲染顺序相邻且材质相同的SubMesh则会动态将其索引数组拼合，从而合成一个Draw Call。而Unity 5.3之后则不再拼合索引数组，因为在不切换材质时产生多个Draw Call的开销并不大，而这多个Draw Call会被统计为一个Batch。

无法参与或打断的情况：

\*打断：改变renderer.material会造成材质的拷贝，打断批处理。要使用shaderedmaterial来保证材质的共享状态。

\*打断：静态批处理的对象必须按顺序一起渲染！如果渲染顺序在批次之间有其他渲染，那么它将被分解成多个批次！

\*打断：有lightmap的物体含有额外（隐藏）的材质属性，比如：lightmap的便宜和缩放系数等，拥有lightmap的物体将不会进行同批处理（除非他们使用同一张lightmap）

\*分组：合并的mesh顶点太多会分组，大约每组15k？？32-64k？？？

动态合批：

GPU Instance：

静态合批不用考虑模型的Scale正负，但GPUInstance合批时模型，scale三轴正负不同轴数量为奇数时不可合批，有两个轴正负不同时可以合批。

AUP：Async Upload Pipeline

<https://blog.csdn.net/uwa4d/article/details/99414712>