目录

[宏观渲染管线 2](#_Toc69637255)

[应用阶段 2](#_Toc69637256)

[几何阶段 3](#_Toc69637257)

[光栅阶段 4](#_Toc69637258)

[关于照明部分的前言 6](#_Toc69637259)

[一帧由哪些渲染流程组成？ 6](#_Toc69637260)

[从Forward Rendering说起 6](#_Toc69637261)

[Z-Prepass和多遍渲染 7](#_Toc69637262)

[Deferred Rendering 8](#_Toc69637263)

[Tiled Based Deferred Rendering & Forward+ 13](#_Toc69637264)

[Clustered Shading 17](#_Toc69637265)

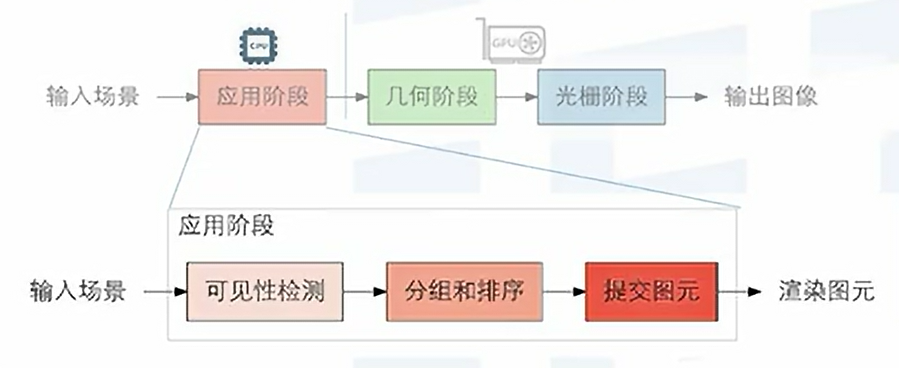
[总结 18](#_Toc69637266)

# 宏观渲染管线



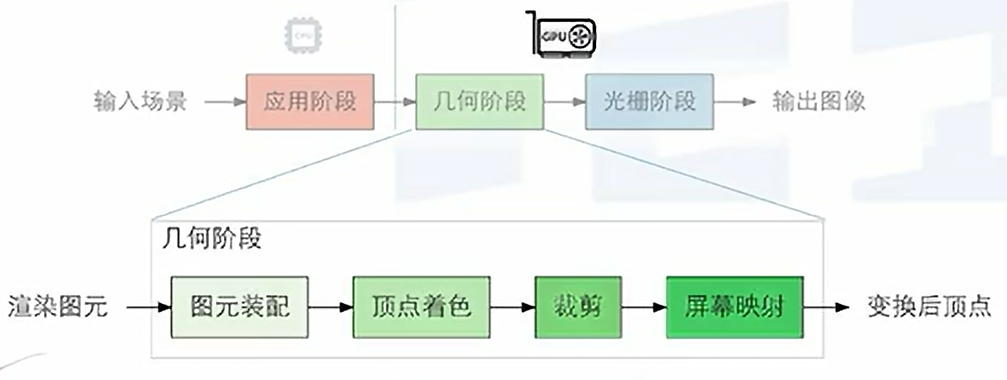
1. 应用阶段：找出场景中需要渲染的物体，将这些物体的图元提交到GPU。
2. 几何阶段：处理和变换图元。
3. 光栅阶段：把几何形状光栅化，计算每个像素的输出颜色。

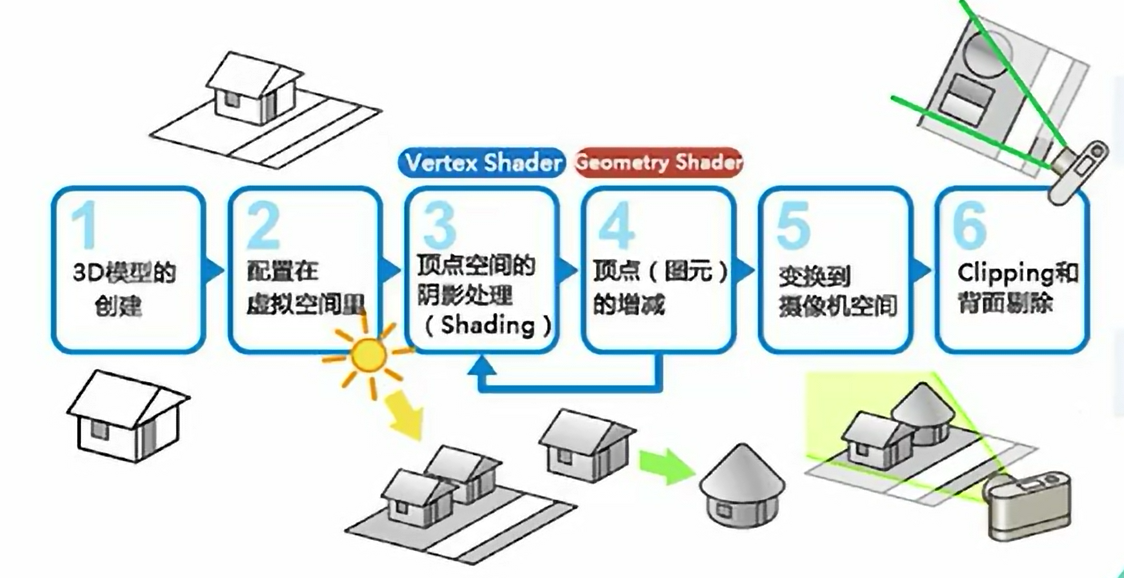
## 应用阶段



1. 可见性检测：在场景管理中执行，通过场景管理中的一些空间分割技术（比如BSP树、八叉树、portal和anti-portal等）可以加速场景的查询。同时可见性检测找出摄像机可以看到的渲染物体，即对之前查询出来的需要渲染的物体进行进一步筛检。其中的方法有：
2. 平截头体剔除：平截头体外的物体都不会被渲染。
3. 遮挡剔除：所有被挡住的物体都不会被渲染。
4. 细节层次剔除：靠近摄像机使用高模，远离摄像机使用低模。
5. 分组和排序：通常按照渲染层分组。半透明物体不能用Z-Buffer排序，要绘制半透明物体可以使用次序无关透明技术（OIT）。
6. 提交图元：提交图元（primitive）就是所谓的DrawCall，由应用阶段输出数据给接下来的阶段。提交前需要设置图元数据和渲染状态。

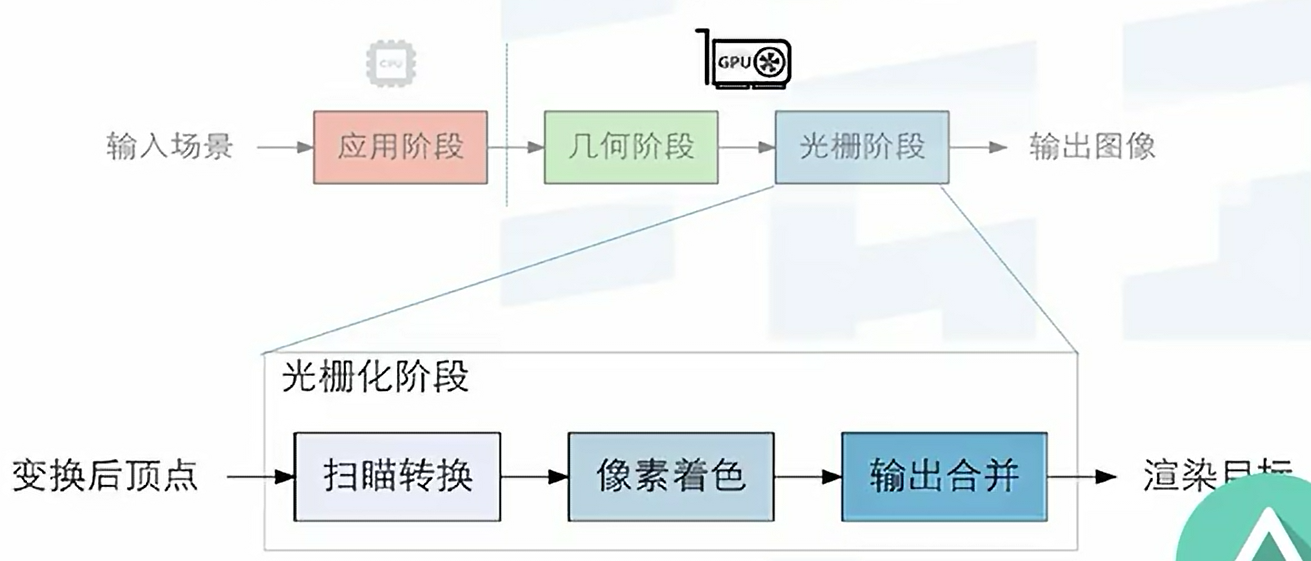
## 几何阶段



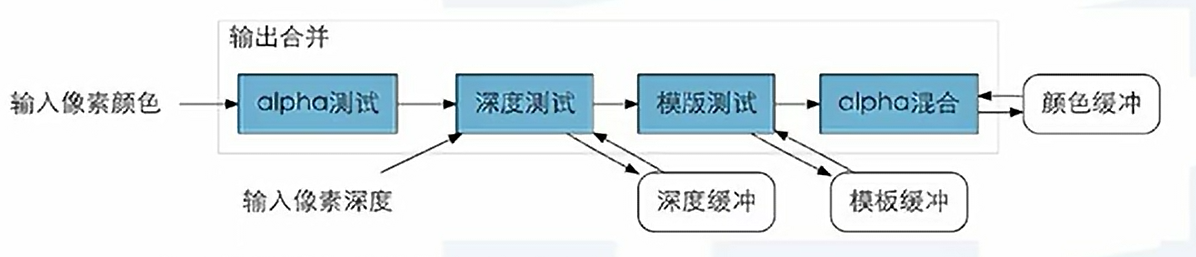


1. 输入装配：根据图元的拓扑类型及顶点格式读取VB和IB，生成三角形的顶点数据。
2. 顶点着色：对每一个顶点执行顶点着色器。输入一个顶点的属性，计算变换、纹理坐标、顶点光照等内容，输出顶点的裁切空间坐标（必须）、纹理坐标（可选）等信息。
3. 几何图元的顶点通常定义于模型空间，经过一连串变换，得到屏幕空间的坐标：1. 模型空间。2. 世界空间。3. 观察空间。4. 齐次坐标空间。5. 屏幕空间。
4. 顶点着色可以实现的效果：如上所述几何阶段的大部分计算都是对于顶点位置相关的计算，在顶点着色器中可以实现顶点位移相关的效果，如顶点动画、骨骼蒙皮动画、平面阴影等效果。并且所有顶点着色计算是对每个顶点的，所以可以实现纹理的坐标动画和顶点光照的计算。
5. 几何着色器（可选）：几何着色器以一个或多个表示为一个单独基本图形的顶点作为输入，比如可以是一个点或者三角形。几何着色器在将这些顶点发送到下一个着色阶段之前，可以将这些顶点转变为它认为合适的内容。
6. 朝向剔除：缠绕顺序决定面的正反面，朝向剔除时可以选择剔除屏幕空间中的cw（clockwise）的面或者ccw（counter-clockwise）的面，即不对物体的背面进行渲染。Unity默认将cw设为正面，ccw设为反面。
7. 三角形裁剪：三角形在齐次空间中超出的部分被进行裁剪，裁剪掉超出六个裁切平面的部分，余下部分可能要分割成多个三角形。裁剪计算在齐次裁剪空间中的[-1, +1]范围内进行计算。
8. 透视除法和视口变换：透视除法（Perspective Division）把齐次坐标转换成归一化的设备坐标（NDC，Normalize Device Coordinate）。最后，把NDC坐标转换至视口（viewport），输出按像素为单位的坐标。

## 光栅阶段



1. 扫描转换：扫描转换（Scan Conversion）把三角形数据转换成像素的位置，除输出屏幕坐标，需要对颜色、纹理坐标进行插值（须透视正确）。出现锯齿就是在这一步。
2. 像素着色：在每个像素位置上，执行可编程的像素着色器（Pixel / Fragment Shader），这些计算一般是计算最后输出的颜色，比如对纹理采样、光照计算、阴影计算、最后输出颜色的修正等。
3. 输出合并：



管道的最后是写入渲染目标（Render Target），可以选择是否写入颜色缓冲、深度缓冲、模板缓冲。

# 关于照明部分的前言

渲染管线是做图形的同学最常听到的名词，它描述了游戏中的一帧渲染的流程顺序，有关渲染的所有环节可以说都被囊括其中。对整个渲染流程感兴趣的可以看[HOW UNREAL RENDERS A FRAME](https://interplayoflight.wordpress.com/2017/10/25/how-unreal-renders-a-frame/)和[Leveraging Ray Tracing Hardware Acceleration In Unity](https://auzaiffe.files.wordpress.com/2019/05/digital-dragons-leveraging-ray-tracing-hardware-acceleration-in-unity.pdf)，也可以看[GTA V - Graphics Study](http://www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study/)和[Reverse engineering the rendering of The Witcher 3](https://astralcode.blogspot.com/2018/11/reverse-engineering-rendering-of.html)的帧分析。总体来说渲染管线的组成大同小异，这里主要关注直接照明部分的管线差异。

# 一帧由哪些渲染流程组成？

不管是什么样的渲染管线，它们的目的都是尽可能地呈现真实世界中的各类材质，从这个目的出发，一帧渲染要完成的任务可以用这张图比较简单的解释：



实际上，绝大部分渲染引擎要解决的无非就是**直接照明、间接照明和后处理**。在现代的实时渲染引擎中，直接照明也往往是管线的核心。尽管许多人大概已经听腻了Forward Rendering和Deferred Rendering，也大概知道它们的思路和差异，但你真的了解这些管线实现中的每个细节吗？

## 从Forward Rendering说起

游戏引擎的设计之初，硬件的计算能力还没有那么强，所以能把一个模型的位置画对、有正确的纹理贴图，就已经谢天谢地了，光照？不存在的。所以那个年代的渲染，也基本不区分直接、间接照明，无非就是一堆trick怼上去，弄几个参数调一调，在一个shader里搞定就好了。后来大家渐渐意识到，这个世界要有光，于是逐渐迭代出了我们如今比较流行的几种理想光源。但是受限于Shader Core的运算性能，一个场景能够渲染的光源仍然有限，每个物体通常只有一盏方向光（全局，用于模拟太阳光）和两盏局部光（点光、聚光灯）的预算，这种情况下，大家能够想到最直观的方案就是扩展原本的shader，把光源数目写死，这种方案就是我们熟悉的**Uber Shader**：所有事情都放在一个shader里做完。

Uber Shader直到今天，在一些性能受限的平台上，仍然是主流方案，比如绝大部分WebGL和手机平台上的引擎，都采用这样的方案。这个方案有一些明显的优点，比如**每个模型只绘制一次，每种材质可以使用不同的光照模型和渲染技术**（比如很容易同时实现卡通渲染、各向异性材质和普通材质），缺点当然也很明显，就是**光源数目受限**。

在具体实现的时候，Uber Shader通常会根据光源的数目，编译出几个不同的shader来，然后在CPU端提交渲染前，根据场景内灯光的位置半径以及被渲染物体的包围盒，粗略地算出所有影响它的光源，并根据光源的数目选取不同的预编译好的shader（超过最大数目，则使用最近的N个光源）。

## Z-Prepass和多遍渲染

随着硬件性能的增强和画质需求的提高，固定数量的光源再也无法满足实际应用的需求；同时由于Depth Test的存在，大量在Uber Shader里被绘制的物体，其实都无法被真正的看到，也就带来了性能浪费。有没有更灵活，同时性能又更好的计算方案呢？大家发现，可以**借助当时大部分硬件都有的Early-Z特性，来避免overdraw的问题**。具体来说，在实际渲染之前，加入了一个称之为Z-Prepass的流程，这个流程关闭了Color Buffer的写入，同时Pixel Shader极为简单或者索性为空，可以非常快速的执行完毕并且获得场景中的Z-Buffer；紧接着，我们再关闭Z-Buffer的写入，改depth test function为equal。这样整个渲染过程中，那些没有通过Depth Test的像素，就不必再执行Pixel Shader的复杂运算和写入，对于overdraw比较大的应用来说，能够大大减少Pixel Shader的运算和带宽开销，所以已经几乎成为了渲染管线中的标配。

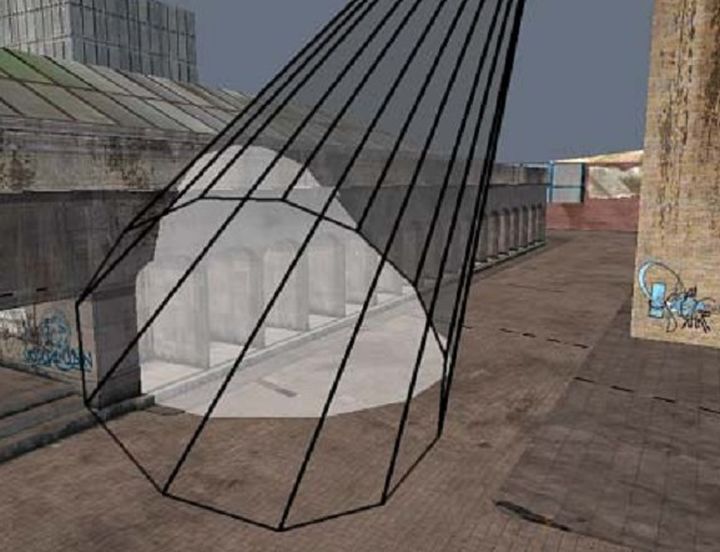
有了Z-Prepass之后，人们就自然而然的想到了支持多光源的方案：**多遍绘制**。几盏光源就绘制几次物体，由于Early-Z的存在，那些不是最终可见的像素并不会实际绘制，所以从性能的角度来讲尚可，灵活度上也大大提高，甚至我们可以结合Uber Shader和多遍绘制，在光源数目较多的时候，减少DrawCall的同时又不失灵活度（比如每遍计算四盏光源而不是一盏）。

实际上Z-Prepass并不总是带来性能的提升，因为它本身会使得Vertex Shader的执行以及DrawCall次数翻倍。[Z-Prepass Considered Irrelevant](http://casual-effects.blogspot.com/2013/08/z-prepass-considered-irrelevant.html)就比较详尽地分析了这个问题。

## Deferred Rendering

到多遍绘制为止，理论上我们已经可以支持无限数量的光源。但即使是结合Uber Shader，场景内的DrawCall数量也是随着光源数目的增加，呈的复杂度（m表示场景内物体数量，n表示灯光数量）。而现代游戏画质的提升，其中一个很重要的指标就是局部光源数目的增加，当场景内有数百个光源的时候，多遍绘制也无能为力了。此外，多遍绘制的另外一个缺陷是，**它对光源的剔除是逐对象的**，这就意味着即使某个光源只照射到物体很小一部分，也仍然需要一次单独的DrawCall。有没有办法把物体的绘制和光照计算解耦开呢？在这样的需求下，得益于MRT（Multiple Render Targets，允许应用程序一次渲染到多个缓冲区）技术的支持，诞生了延迟渲染。它的核心是G-Buffer，也就是若干张贴图，存的是计算光照需要的所有参数，通常包含了depth、normal、albedo、roughness、specular和metallic。

传统的延迟渲染在G-Buffer生成之后，会根据光源的形状（Light Volume），对每个光源执行一次DrawCall，如果某个像素被Light Volume覆盖到了，我们就在该像素的位置执行一次当前光源的Lighting计算。由于光照计算是线性可叠加的，所以我们只要把Color Buffer的Blend Mode设置为ADD，并将Src Factor和Dst Factor设置为ONE即可。

（Light Volume）

需要注意的是，为了防止同一像素被光源正反面计算两次，我们需要在绘制Light Volume的时候使用单面渲染，如果摄像机在光源内，则需要开启正面剔除，并且将Depth Test设置为farOrEqual，如果摄像机在光源之外，则开启背面剔除，并且将Depth Test设置为nearOrEqual。

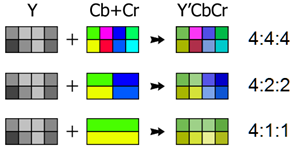
Deferred Rendering的优点是，解耦了Mesh Draw和Light Draw，保证物体只被绘制一次，光源也只绘制一次，场景的DrawCall复杂度变成了。另外，G-Buffer除了用于直接照明外，还能够被用于一些间接照明的效果，比如SSAO、SSR；也正是G-Buffer概念的提出，使得近十年来越来越多的算法从World Space向Screen Space的演进；去年因为硬件加速而流行的Ray-Tracing技术，得益于G-Buffer，也能够大大提高性能。延迟渲染的缺点主要在于带宽：G-Buffer要存的数据越多，带宽开销就越大。

带宽的优化来自于两个方面：**读和写**。写的部分主要是G-Buffer的压缩，在这个方向演化出了许多用于压缩和减小G-Buffer的方案，比如早年Crytek提出的[CryEngine3: Reaching the speed of light](http://advances.realtimerendering.com/s2010/Kaplanyan-CryEngine3%28SIGGRAPH%202010%20Advanced%20RealTime%20Rendering%20Course%29.pdf)和[其他一些normal的压缩方案](https://aras-p.info/texts/CompactNormalStorage.html)，以及[基于YCbCr（或者YCoCg）的色彩空间把三通道的RGB信息压缩到两通道](https://www.slideshare.net/TiagoAlexSousa/rendering-technologies-from-crysis-3-gdc-2013)（当然色彩压缩的方案由于最终着色的时候还需要多读一个pixel的值去重建，所以得失如何还不好说）。G-Buffer的压缩方案一个缺陷是，**它可能会导致硬件的Blend Mode失效（主要影响Decal Blend）**。



Crysis3用的Thin G-Buffer





YcbCr的压缩方式

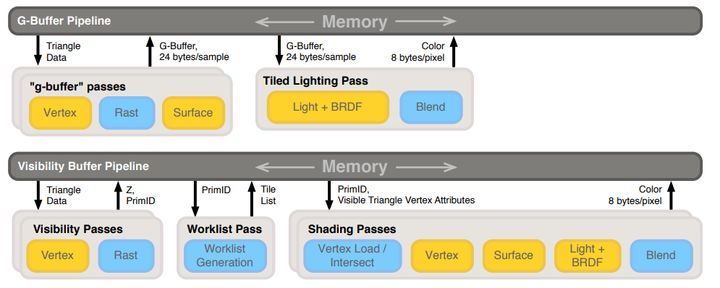
另外一个优化带宽的方法称之为**Deferred Lighting**，它的思路是，进一步解耦光照和最终着色的流程，我们以一个简单的Blinn-Phong的Shading Model为例，它的着色计算可以用这个公式描述：

对于k个光源，我们需要读k次albedo和specular的值，但实际上，对同一个像素来说，和作为常数可以提取出来，于是方程改成如下形式：

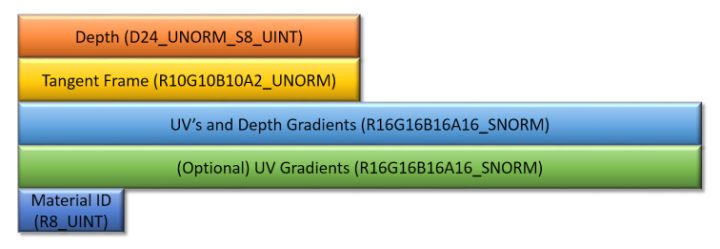
于是我们在Lighting时分别存储和，之后再加一个Shading的流程，在Shading时读取和的值，这样，albedo和specular就只需要读取一次，减少了带宽；相应地，我们需要存储两张Lighting Buffer。当然，我们也可以只存储RGB（Diffuse）和Lum（Specular），然后用

来恢复，以节约Lighting Buffer的写入带宽。此外，当我们使用菲涅尔项去更好的模拟specular的时候，Deferred Lighting又变得不可行了，这时候我们只能把近似为。有关Deferred Lighting的更多细节，可以参考[deferred lighting approaches](http://www.realtimerendering.com/blog/deferred-lighting-approaches/)和[light-prepass](https://www.slideshare.net/cagetu/light-prepass)。

[The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading](http://jcgt.org/published/0002/02/04/paper.pdf)和[BINDLESS TEXTURING FOR DEFERRED RENDERING AND DECALS](https://mynameismjp.wordpress.com/2016/03/25/bindless-texturing-for-deferred-rendering-and-decals/)是在Deferred Lighting基础上更为激进的方案，它的思路很简单：既然G-Buffer这么费，我们索性不渲染G-Buffer，改为渲染Visibility Buffer，这个buffer上只存primitive ID、uv和贴图ID，我们根据这些属性，分别从UAV和Bindless Texture里面读取我们Shading真正需要的Vertex Attributes和贴图的属性，根据uv的差分自行计算mipmap：



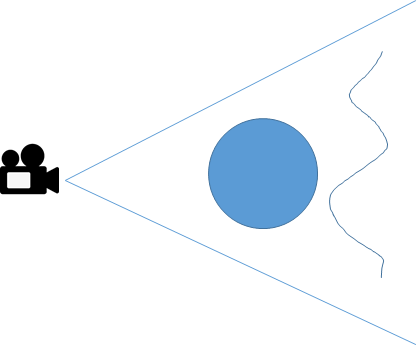
传统G-Buffer和Visibility Buffer在渲染流程上的差异



Visibility Buffer的layout

这个方法确实能够一定程度上减少G-Buffer的容量（所有材质相关的属性都不存了），但是它的缺点是需要Bindless Texture的支持，而且由于贴图是在Screen Space根据Material ID动态索引的，所以从缓存友好的角度来说，我觉得不见得好（毕竟像素之间可能出现贴图跳变）。

除了上述的方案去优化读写带宽，针对传统的Deferred Rendering，还有一些方法是旨在减少Light Volume中无效的像素。我们之前已经描述了Light Volume的方法，但是，试想这种情况：



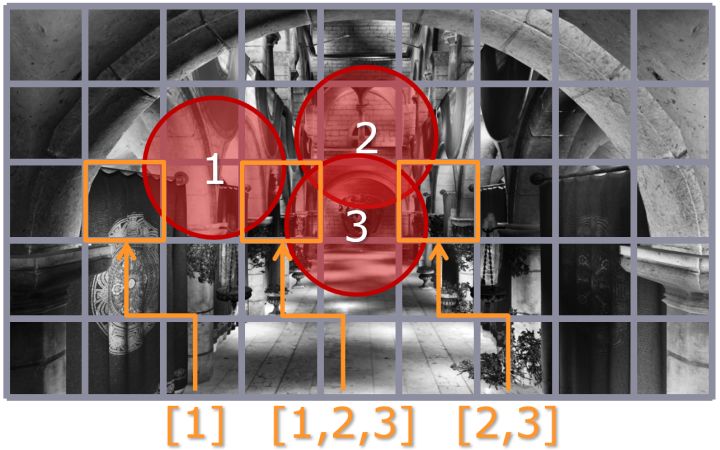
当光源整个位于Z-Buffer前面时，整个Light Volume的区域都不需要实际渲染

这种情况下，虽然整个光源的正面都通过了NearOrEqual的测试，但其实它的所有区域都不对场景产生光照贡献。针对这种情况，有两种优化方案：

1. **Depth Bounds Test**：这个方案利用了硬件中的一个称之为Depth Bounds Test的特性，在普通Depth Test之外，额外增加一次测试，这个测试从Depth Buffer里面读取深度，然后用这个深度和Depth Bound Test设置的zmin和zmax值作比较，如果depth值没有落在[zmin, zmax]这个区间内，则直接拒绝这个像素不执行Pixel Shader。利用这个优化，我们可以在绘制每个光源的时候，计算它的Depth Bound，并设置给管线，这样，上图的情况就不会实际着色。[这个方法在killzone2中曾被使用](https://www.guerrilla-games.com/read/the-rendering-technology-of-killzone-2)，cryengine也用过类似的优化。
2. **2-Pass Stencil Culling**：这个方案剔除更精确，缺点是每个光源要绘制两遍：我们需要真正着色的像素，其实是深度值小于Light Volume背面，大于Light Volume正面的那些位置。所以我们先绘制Light Volume的正面，Pixel Shader置为空，将NearOrEqual的像素用Stencil Buffer标记出来；然后绘制Light Volume的背面，Pixel Shader设置为光照的Shader，Depth Test设置为FarOrEqual，并且只有被上一个pass标记的像素才通过Stencil Test，这样一来，就能够精确标记并减少Light Shader的浪费，[在killzone2中也有使用](https://www.guerrilla-games.com/read/the-rendering-technology-of-killzone-2)。另外，[Vulkan Multipass mobile deferred done right](https://www.khronos.org/assets/uploads/developers/library/2017-khronos-uk-vulkanised/003-Vulkan-Multipass_May17.pdf)提到了一种**clustered stencil culling**的方法，能够一次绘制多个光源的Stencil，结合Instance Rendering，能够进一步减少2-Pass Stencil Culling的状态切换开销。

## Tiled Based Deferred Rendering & Forward+

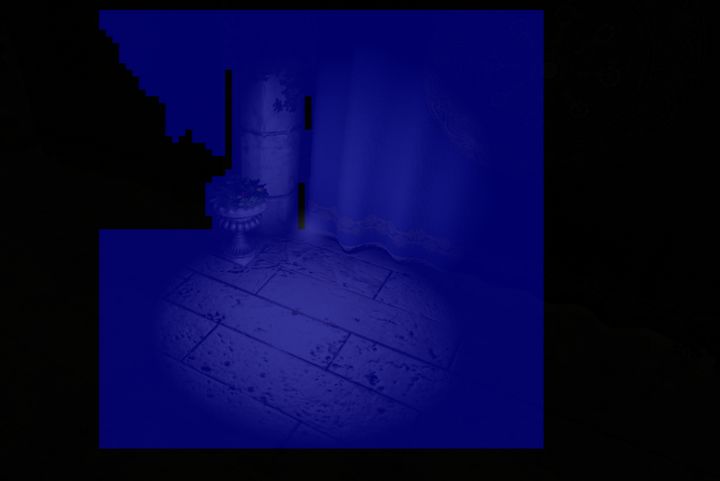
在上述方法之后，工业界开始继续寻找解耦光照和几何计算的方法。在上述方法的基础之上，发展出的就是**Tiled Based**方法。实际上TBDR和Forward+是Tiled Based方法在Forward和Deferred上各自的体现，相较于过去的管线，Tiled Based的方法增加了一个**Light Culling**的流程，这个流程把整个屏幕分割成若干个Tile（通常每个Tile是16\*16个Pixel），每个Tile各自计算出一个单独的Light List，找出场景中那些对当前Tile有贡献的光源。然后对每个Tile中的Pixel，只需要计算其对应的Tile中Light List内的光源对该像素的贡献。

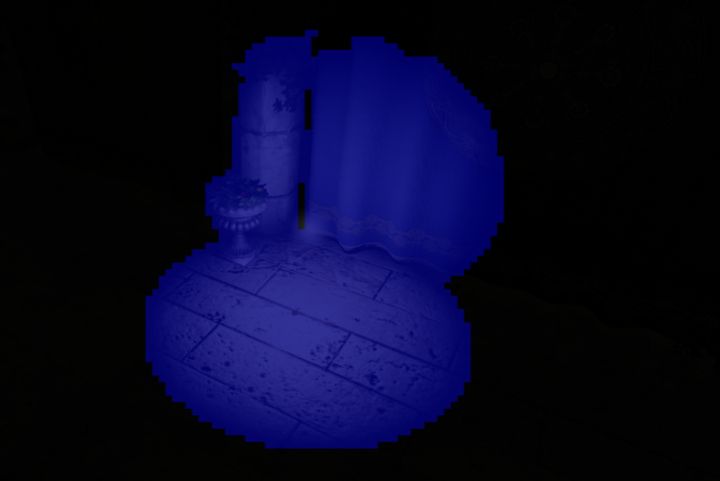


Light Culling需要一个额外的Compute Shader来计算，依据是当前Tile的Screen Positon和Depth Bound（基于这两个信息可以计算出View Frustum）。**对于Forward方法来说，是在Z-Prepass之后，而对Deferred方法来说，是在G-Buffer pass之后**。每个group内单独地对场景所有光源进行相交测试，每次可以并行测试256盏光源（16\*16）。对于Forward来说，计算出来的Light List需要存储在一个UAV Buffer中，然后再执行一次Geometry Pass，根据每个Pixel所在的Tile去索引UAV Buffer中的Light List，然后用动态的循环去计算每个光源对当前Pixel的贡献；而对于Deferred来说，Light List可以存储在Compute Group对应的Shared Memory中（因为Light Culling就是以Compute Group为单位去运行的），Light Culling之后，在同一个Compute Shader内执行光照计算。

Tiled Based方法可以说进一步减少了带宽，因为对于一个像素来说，相较于传统Forward / Deferred方法，涉及光照计算的Pixel Shader只执行一次，也就意味着**Material Data（比如albedo，normal，specular，roughness，metallic等）都只需要读取一次，并且最终结果也是一次写入**。但是它也有比较明显的缺陷：相较于传统的Deferred Shading，TBDR的方法对于光源的剔除更粗粒度了（**逐Tile而不是逐Pixel的**），所以**当光源数量不够多的时候，这个方法并没有明显的性能优势**。

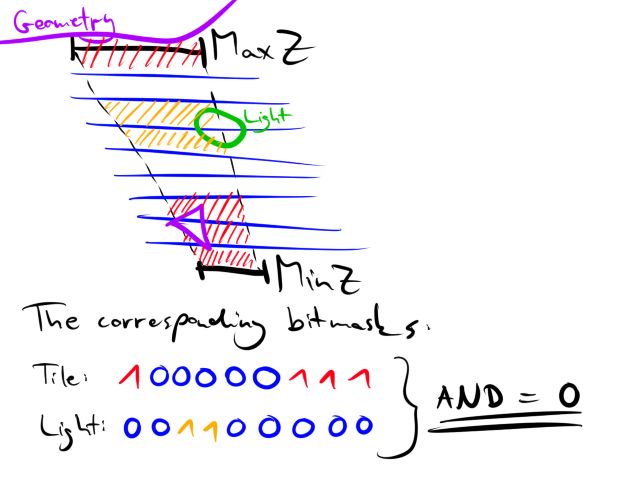
Tiled Based基本的剔除方法是基于Frustum的，它的缺陷在于，**基于Frustum Culling的方案对场景光源的剔除不够精确**，这也是它在较少光源时性能不如传统Deferred的原因之一。针对这个问题，很多方案提出了更精确的相交测试的方案，比如结合Depth Bound算出当前Tile的AABB，然后基于AABB和Bounding Sphere做相交测试（[Optimizing-tile-based-light-culling](https://wickedengine.net/2018/01/10/optimizing-tile-based-light-culling/)）。





基于Frustum Plane的剔除和AABB剔除的结果差异，可以看出AABB的方法大大减少了无效的Tile

另一个相交测试不准的原因是，场景里**每个Tile的Depth的分布并不是连续的**，在[zmin, zmax]区间内的很多深度范围可能是没有像素的，而光源可能正好分布在这个区间内。对于这个问题，分别有两个解决方案，其一是[**Depth Split**](https://www.gdcvault.com/play/1017627/Advanced-Visual-Effects-with-DirectX)，简单说就是把一个Tile根据[zmin, zmax]再一分为二，分别计算每个区间的Light List，另一个方案叫作**[2.5D Culling](https://wickedengine.net/2018/01/10/optimizing-tile-based-light-culling/)**，就是把当前Tile的Depth区间分成32段（刚好是一个uint32），再把Tile里每个像素的Depth映射到其中一段，然后写入一个DepthMask，然后根据Light Bounding Box，也把它的深度映射成一个LightMask，根据(DepthMask & LightMask == 0)来判断一个Tile是否真的和一个Light相交。



2.5D Culling的算法示意图

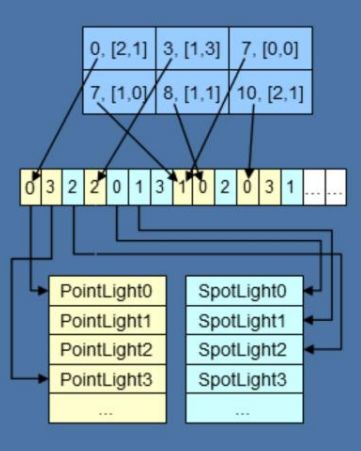
除了一般性的相交测试的准确性问题和Depth Discontinuous的问题之外，[这篇文章](http://twvideo01.ubm-us.net/o1/vault/gdc2016/Presentations/Zhdan_Sjoholm_Light_culling_MGPU.pdf)还提到了结合Light Volume（也就是光栅器）的结果进一步提高Light Culling准确率的方法，感兴趣的同学可以读一读。

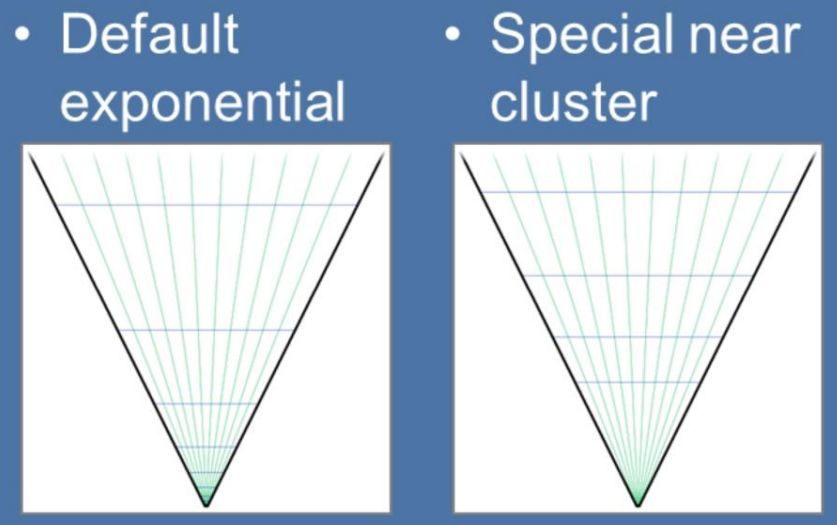
针对Foward+和TBDR，[这篇文章](https://www.gdcvault.com/play/1017627/Advanced-Visual-Effects-with-DirectX)做了一个比较详尽的比较，总体来说结论还比较直观：**当光源并不是真的非常多（超过2048个）时，只有使用MSAA时，Foward+好于TBDR；否则均是TBDR好于Foward+**，鉴于目前各种Screen Space的AA方案的成熟（SMAA，FXAA，TAA等），MSAA实际上已经变得越来越鸡肋，从性能的角度已是一个拖累。加上我们之前说到的G-Buffer带来的额外红利，目前市面上的引擎仍然是主流使用Deferred框架。而针对无法表征多种Shading Model的问题，**目前的Deferred引擎多数采用Shading Model ID + Custom Data的形式，再加上在Lighting时做动态分支的方法来处理**。

## Clustered Shading

尽管我们觉得Tiled Based的方法似乎已经能够比较好的解决多光源的问题（至少场景内几百上千个光源没什么问题），但是对现代的3A大作来说，这仍显得不够。现代3A大作的要求是，场景内要数千乃至上万个光源，鉴于我们已经提出了Light Culling的方案，那我们就需要进一步地思考**如何用更细粒度的数据结构来存储Light List**，基于这个思考，比较直观的结果就是[Clustered Shading](https://newq.net/dl/pub/s2015_practical.pdf)，这个思路给Light List的划分增加了一个维度，即Depth（当然也可以再增加normal的维度），它根据View Frustum的zmin，zmax把场景进一步根据Depth划分成若干个Slice（基于指数的划分，通常16个），然后在每个Slice上对场景中的所有灯光进行Light Culling，具体的计算方案和Tiled Based提到的一些方案类似，只是这里**不再需要处理深度不连续的问题**。所以Slice的Light List被计算出来后，会经过一个Sort和Compact的流程（[基于Compute Shader](http://www.cse.chalmers.se/~uffe/clustered_shading_preprint.pdf)），最终形成的存储结构有三个：

1. 一个3D Texture，用于存储每个Depth Slice上的光源的起始位置和光源数量；
2. 一个UAV Buffer，存储每个光源的具体索引，UAV中的offset和num lights由1决定；
3. 一个Constant Buffer，存储具体的光源数据。

（整个Light List的存储结构）



Depth Slice的一些划分形式

在实际Shading的时候，每个像素根据自己的Depth和Screen Position，找到对应的Depth Slice，从3D Texture里拿到offset和num lights，再执行一个num lights次循环，从offset处取到UAV Buffer的索引，然后从Constant Buffer里拿到具体的光源数据执行光照计算。

Tiled Based和Clustered Based的方法除了能够提高带宽的利用率，另外一个重要的优势是，**它和Forward / Deferred方案是正交的**，这意味着不管你是什么基本管线，都可以使用这个方法处理光源列表，而对于现代引擎来说，**它的渲染管线往往是以Deferred为主，结合Forward处理一些半透明和特殊材质的混合管线**，因此Tiled / Clustered的方案就更显示出了它们的优势。

# 总结

有关本文涉及的部分管线在移动端性能对比，可以参见[Rendering Structures](https://community.arm.com/cfs-file/__key/communityserver-blogs-components-weblogfiles/00-00-00-20-66/5127.siggraph_2D00_2018_2D00_mmg_2D00_3_2D00_rendering_2D00_hanskristian.pdf)。在Siggraph 2017上，[Improved Culling for Tiled and Clustered Rendering](http://advances.realtimerendering.com/s2017/2017_Sig_Improved_Culling_final.pdf)历数了各种管线和光照的优化方案，并且在指令级别和数据结构的角度提出了许多光照剔除优化的方案，也非常建议大家详细阅读。

其实我常常觉得，哪怕是一个简单的技术，哪怕你每天在和它打交道，或许仍有些许内容你是不了解的。就像本文中的前向渲染和延迟渲染，很难说有那个图形工程师对此没有了解。然而有时候，当你从头梳理这项基本技术的时候，还是会发现许多前人为此付出的努力，并且许多看似过时的技术，当你清楚地理解了它产生的原因和解决的问题后，某些思路仍然能够迁移到其他问题上。