延迟渲染

我一直认为，延迟渲染跟阴影，是图形学的一个分水岭。能深刻的理解了延迟渲染、阴影，那么至少图形学算是入门了，至少对渲染架构，渲染的一些基本算法，入门了，不再是门外汉了。

声明一下，这里只打算讲原理，并且实现一些伪代码。这东西大行其道都有十年了吧？要抄代码的话，网上到处都是，没必要在这里抄。况且，DX11，DX9，GL，Vulkan的实现都不同，理解了原理，搞搞就出来了，抄的话，一般只是借鉴一下GBuffer的压缩有什么好的法子。

先抛出一个问题：我们为什么需要延迟渲染？主要解决什么问题？一般面试的时候，你问这个问题，应聘的基本上都会有一个差不多的标准答案：主要解决大量光照的渲染消耗。

延迟渲染有什么不足？一般也会有过得去的答案：1、不能用硬件AA（抗锯齿）。2、透明渲染不行。

能够这么回答的，应该都看过了，有准备的。这个时候，要判断是不是有真功夫，就要问更深入的了。例如可以直接问，除了这两个不足，还有其他的吗？这个时候，如果直接说没有的，那么这个就是教科书答案。如果能回答出：占用了大量的显存带宽的，那么显然水平更高一筹，估计是实战过的，又或者看得更多。如果还能更深入的说出：延迟渲染只能用同一套lighting pass，那么，要么是取巧到了极致，要么是大概率有真才实学的了。

为什么需要延迟渲染？我们来看看传统的渲染是怎么做的：

假设有10000个Mesh需要渲染，有100盏灯，大概需要这么做：

For(int i = 0 ;i < 10000; i++)

{

For(int j = 0; j < 100; j++)

{

RenderMesh(MeshData, Light);

}

}

而RenderMesh这个函数里，大概需要这样：

Void RenderMesh()

{

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

{

RenderTriange(Light);

}

}

看到了吗，这个恐怖的计算量，GPU也顶不住啊。

我们看到，每一个像素渲染的时候，都需要跟灯光做一个着色计算，计算灯光是否照到了这个像素，照到的话，什么颜色。这里面，其实绝大多数，都是无效计算。为什么？

因为除了平行光，绝大部分光，例如点光源，聚光灯，范围都不会太大，照到的像素有限，照不到的，也计算一遍，就成了无效计算了。

你总不能100个都是平行光吧？那得多脑残才能设计出这样的场景啊，我是没见过。

总结一下：如果你的场景只有一个平行光，那么使用延迟渲染优化将毫无意义！图形学的很多优化算法，都是针对特定的场景，而不是一个算法吃遍天，这不现实。例如GIS的渲染跟游戏的渲染，游戏无限地形渲染跟小地图渲染，诸如此类，往往优化方式都是不同的。

用延迟渲染，是如何减少无效计算的？

回到之前的代码，假设TriangleCount每个mesh都是一样的，都是1000个。而每个三角形在屏幕上的像素也是一样的，都是1000个，那么总共进行了多少次计算？大概是：

1000（像素） \* 1000（三角形） \* 100 \* 10000 = 10^12那么多次计算。这种计算量，相当恐怖了，每秒钟还能渲染多少帧？

延迟渲染呢？延迟渲染的意思是：先确定渲染到屏幕上的像素，再用像素去做光照。渲染的流程大概是这样的：

For(int i = 0 ;i < 10000; i++)

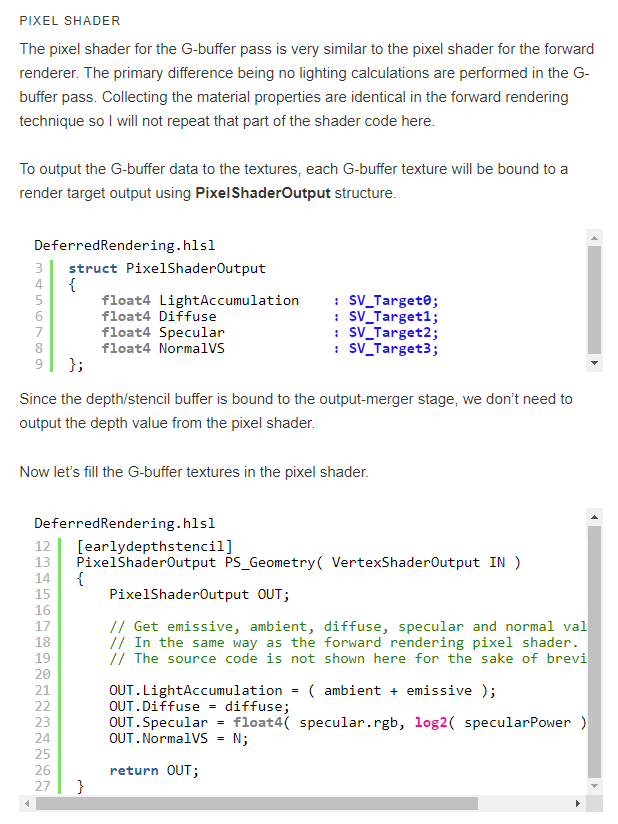
{

RenderMesh(MeshData, Light);

}

RenderMesh的时候，不是直接计算光照，而是先把一些必须的信息保存下来，例如可以保存世界坐标。保存法线，保存采样贴图颜色……这个，就是所谓的GBuffer。一般这个用MRT来实现，Multi Render Target。一般来说，这里至少需要两个Target，我之前写的时候用了三个。我听说过七个的奇葩，那这种消耗也是太离谱了。

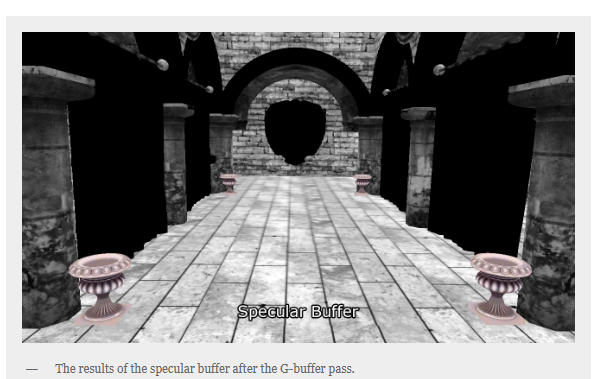
看看这个Gbuffer，用了四个贴图：



上图直接从网络截取，不找自己的代码了。

得到了GBuffer，再计算光照。由于屏幕上的像素就那么多，例如1024 \* 768个，你只需要把这些像素点全部进行光照迭代就可以了。

看图，要切记，不管场景中有多少个模型，有多少的三角形，你能看到的，永远是渲染窗口的像素！



所以，延迟渲染的实质，是先不要做迭代三角形，做光照计算，而是先找出来你能看到的所有像素，再去迭代光照。直接迭代三角形的话，由于大量三角形你是看不到的，无疑是极大的浪费。

所以，看完了延迟渲染的原理，延迟渲染的缺陷，很容易看懂了。我们来一个一个回顾这些缺陷是如何产生的。

1. 不能使用硬件AA（MSAA），这个，我在其他地方应该讲过，render to texture，是不能用MSAA的，这个跟AA的原理有关。所以使用了延迟渲染之后，UE4只支持FXAA跟TXAA。
2. 不支持透明物体的渲染。为什么呢？很简单啊，因为之前渲染Mesh的时候，是一个一个迭代的，一个一个做深度测试之类的，但是，延迟渲染要先渲染到Gbuffer。可想而知，Gbuffer只是把当前能看到的像素记录下来，但是透明的，同一个像素点，可能需要记录更多！所以，延迟渲染的时候，一般都是先渲染非透明的Mesh，后续再单独渲染透明Mesh。
3. 占用大量的显存带宽。这个其实也好理解，说穿了，就是显存的读写，是非常受限的，我应该在“认识显卡”章节，有单独介绍过显存位宽的计算。现在主流DDR5显卡，带宽大概是200GBit好像，假设渲染窗口是1920 \* 1080 \* 4（RGBA）\* 4（MRT）\* 8（Byte to bit） \* 60（FPS），那么，光是这个Gbuffer，占用的带宽，我算了一下，就是15G。这其实已经是极大的消耗，在台式机还能勉强承受，在移动端设备，我估计就这个就坑逼了。
4. 只能使用同一个光照pass。这个很好理解，因为如果是一个一个mesh迭代，我一个mesh用一个material，里面用自己的光照算法。另外一个mesh我不想用光照，直接一个白板pass，当然是可以的。但是，渲染到Gbuffer之后，你其实已经不知道哪个像素点属于哪个Mesh了，自然就只能使用同一套光照算法了。这个为什么一般的书里都不提呢，主要是大多数时候，本来一个场景的光照算法就是要统一的，不统一的反而是少数。能理解到这一步，证明至少是深入思考过的。

延迟渲染在写入Gbuffer的时候，有不少的渲染技巧，尽量减少Gbuffer的贴图占用，这里就不一一细说了，那个属于《延迟渲染实战技巧大全》，不属于本书范畴。

延迟渲染还有一个优势，就是所谓的结合SSAO。这个，就不讨论了，有兴趣的，可以去找找SSAO的资料看看，应该烂大街了。

讲了延迟渲染，这里需要顺带讲一下Forward+。延迟渲染大行其道的时候，各大引擎都加入了支持，包括UE，U3D。后来，Forward+出现了，然后到处鼓吹这个效率多好多好，延迟渲染要淘汰了。但是据我观察，各大引擎依然还是延迟渲染，并没有看到什么引擎彻底抛弃了延迟渲染而选择了Fordward+，也许是我孤陋寡闻？

这里，我觉得有必要介绍一下forward+，所谓没有对比就没有伤害。那么，就简单介绍一下吧。先声明，延迟渲染我是自己实操过的，所以理解应该是深刻的。但是forward+我只是看过文档，加入了自己的理解，并没有实操过，有错的话，直接开喷。

跟延迟渲染一样，forward+同样在解决一个问题：大量的实时光照导致场景的重复绘制。如果你很少用灯，或者大量的光照你使用了烘焙，而不是实时计算，那么洗洗睡吧，不需要那么复杂，直接forward渲染即可。

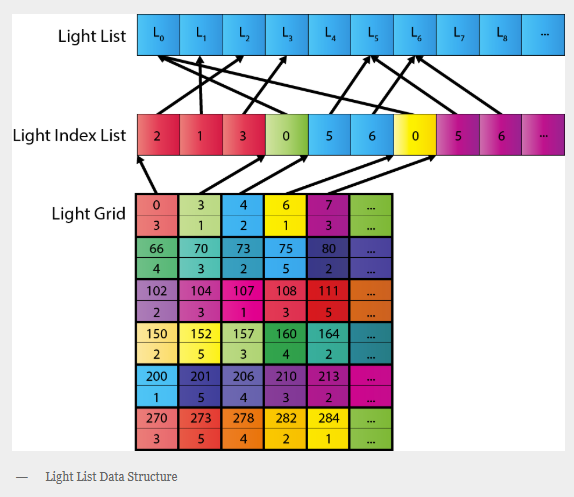
延迟渲染是通过先算出需要着色的像素，然后再迭代灯光，从而减少大量无效的灯光计算，来达到优化的目的。Forward+是怎么做的呢？我们先来看一张大量灯光的图：



假设要渲染一个场景，路灯非常多，你希望路灯都是实时灯，这个场景，是理性的。类似的场景，都有一个特点：灯的范围不会很大！也就是说，每盏路灯，可能就是一个3米、5米的点光源。所以，按照传统的forward+，每个Mesh全部遍历一次所有灯光，是非常不合理的。

Forward+要做的是，把屏幕做一个细分，英文一般俗称Tiled或者grid。例如，我们可以16 \* 16个像素划分为一个小格子，计算这个格子被多少盏灯照到，渲染这个格子的时候，只需要迭代计算到的灯光，而不是需要迭代所有的灯光。假设一个场景有100个实时灯，那么全屏能分开几千个格子，那么，分到一个格子的灯，估计也就那么三五个，完全可以接受了。这样，既能够享受forward渲染的好处，不需要忍受延迟渲染的各种不爽，效率也上来了，岂不是美哉。

这里，核心的算法有两步：第一步，是把相机划分成N个锥体，N等于划分的格子数，一个格子是一个锥体，一般叫做frustum culling。第二步，做light culling，其实就是计算一个灯是否在这个格子里。先把场景的灯放进一个light list，然后每个格子的灯，用light index list，记录灯的索引，大概这样：



图片来自网络。

这两步的实现，是用compute shader来做的，需要使用到不小的计算，显存估计也要用好几M，来保存light index list。

以上，应该是早期的forward+渲染版本，后续应该出了不少的优化算法。

我所知道的，有一种方案，是对前面的frustum culling进行优化的。Frustum culling，有一个很大的问题，是遮挡以及镜头的深度。

假设一个朝大场景，做GIS，一个大城市，镜头能看到5000米，这种情况下，其实绝大部分灯光，是会被遮挡的，这会导致之前的frustum culling造成大量的浪费。看图：



像这种城市俯瞰图，距离假设camera的far是5000米，你这么搞，后面的灯能看清吗？或者前面一栋大楼挡一下，你后面的灯不都是白费了吗？所以，frustum culling还可以做优化。如果你是游戏场景，渲染距离可能也就是三五百米，遮挡很少，那这个优化其实没多大必要。所以，还是那句，3D渲染的优化，大部分都是根据实际情况来的，很难有一套算法打天下的时候。

优化的方案是怎么做的？我知道一套，是用depth来做的。也就是说，你渲染grid的时候，其实不是用camera的视锥体来算culling，而是根据当前grid的像素深度，先算出来一个frustum。这个其实很合理，意思是不在渲染范围内的，不加入计算，尽量减少无效计算。甚至在超大场景的时候，你可以忽略远处的灯光，不要取camera的far，而是另外自己定义一个。因为太远处的灯光，是否参与计算，你其实不怎么在意，也不怎么看得出来。

好了，渲染架构相关的就大概讲到这里。从架构上来说，两个都比较复杂，延迟渲染带来了大量带宽的消耗，导致移动端基本没法用（移动端其实本来就没有大量实时灯的渲染要求，不现实）。而各种对比里面，结论都是forward+效率比deferred略高，因此，早几年出现了一大波鼓吹forward+的热潮，好像deferred已经是明日黄花，已经是obsolete。

然而，这几年并没有发现主流引擎淘汰了延迟渲染的现象。就我自己的看法，我也不觉得forward+真的优秀到了碾压了deferred。

先说AA，其实这两年，TXAA广泛应用，真的不觉得MSAA无可替代，MSAA其实局限性太大了。想象一下，那么多render target，全部不能用MSAA，这本身就导致MSAA非常受限，大家都迫切需要一套render target能AA的算法。所以，不用MSAA其实没有那么大不了的，MSAA消耗还非常大。

再说透明物体的渲染。老实说，这世界，哪里有那么多透明物体需要渲染呢，除非你做一款游戏，以玻璃为背景板，否则，一般的写实游戏，这个半透明并没有起到决定性的作用。实际应用中，我估计粒子，水面这些，占了半透明的绝大部分。粒子的优化方案，就更多了，貌似GPU GEMS里还有介绍一种优化，是先把粒子渲染到小纹理，减少计算，然后再根据depth后处理到屏幕上的。这种情况下，粒子都单独处理了，已经脱离了渲染架构之外。所以透明的渲染其实并没有那么多。

从未来的发展来看，4k应该是未来的大方向。Deferred毕竟是精确到像素级的优化方案，4k渲染的时候，固然GBuffer变大了，但是frustum culling跟light culling的计算量增幅更大，高分辨率下forward+的渲染是不是还能保持这个优势，我是存疑的。

人越年轻，越容易有期待。例如我刚毕业那会，什么都不懂，曾经期待intel、nvidia不要挤牙膏，要放大招，一次带来架构的巨变，实现技术的跃迁。越了解技术的细节，越清楚这不过是不切实际的期待。现有的渲染优化，几乎已经做到了极致。就好比图片格式优化，压缩算法优化一样，都是有物理极限的，不可能无限的进行下去，也不可能出现量级的提升。所以，挤牙膏才是这个时代的常态。

现实总是如此无奈。