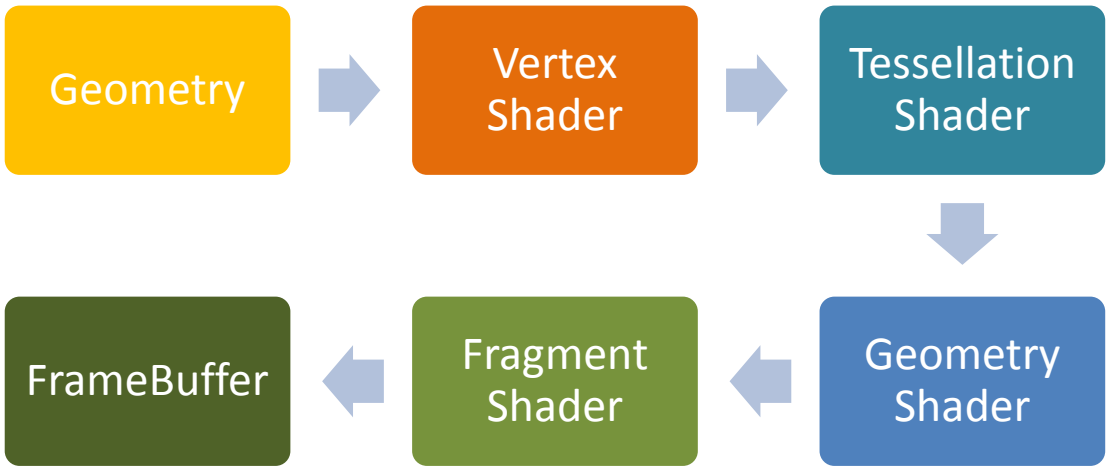
实时渲染中常用的几种 Rendering Path

1. rendering path 的技术基础

在介绍各种光照渲染方式之前，首先必须介绍一下现代的图形渲染管线。这是下面提到的几种 Rendering Path 的技术基础。



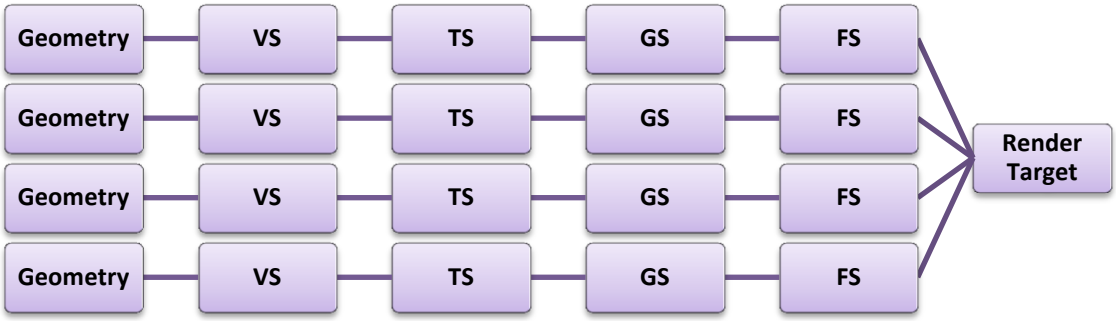
目前主流的游戏和图形渲染引擎，包括底层的 API（如 DirectX 和 OpenGL）都开始支持现代的图形渲染管线。现代的渲染管线也称为可编程管线（Programmable Pipeline），简单点说就是将以前固定管线写死的部分（比如顶点的处理，像素颜色的处理等等）变成在 GPU 上可以进行用户自定义编程的部分，好处就是用户可以自由发挥的空间增大，缺点就是必须用户自己实现很多功能。

下面简单介绍下可编程管线的流程。以 OpenGL 绘制一个三角形举例。首先用户指定三个顶点传给 **Vertex Shader**。然后用户可以选择是否进行 **Tessellation Shader**（曲面细分可能会用到）和 **Geometry Shader**（可以在 GPU 上增删几何信息）。紧接着进行光栅化，再将光栅化后的结果传给 **Fragment Shader** 进行 pixel 级别的处理。最后将处理的像素传给 FrameBuffer 并显示到屏幕上。

2. 几种常用的 Rendering Path

Rendering Path 其实指的就是渲染场景中光照的方式。由于场景中的光源可能很多，甚至是动态的光源。所以怎么在速度和效果上达到一个最好的结果确实很困难。以当今的显卡发展为契机，人们才衍生出了这么多的 Rendering Path 来处理各种光照。

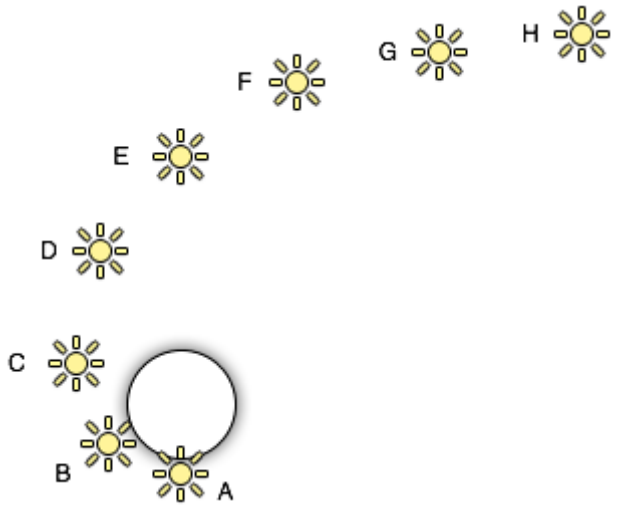
2.1 Forward Rendering



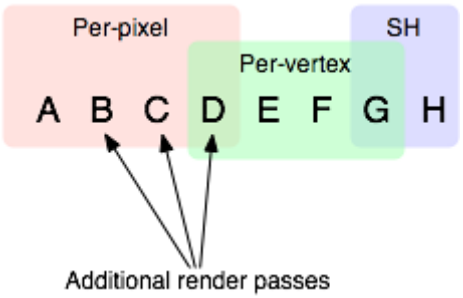
Forward Rendering 是绝大数引擎都含有的一种渲染方式。要使用 Forward Rendering，一般在 Vertex Shader 或 Fragment Shader 阶段对每个顶点或每个像素进行光照计算，并且是对每个光源进行计算产生最终结果。下面是 Forward Rendering 的核心伪代码[1]。

```
For each light:
    For each object affected by the light:
        framebuffer += object * light
```

比如在 Unity3D 4.x 引擎中，对于下图中的圆圈（表示一个 Geometry），进行 Forward Rendering 处理。



将得到下面的处理结果



也就是说，对于 ABCD 四个光源我们在 Fragment Shader 中我们对每个 pixel 处理光照，对于 DEFG 光源我们在 Vertex Shader 中对每个 vertex 处理光照，而对于 GH 光源，我们采用球调和（SH）函数进行处理。

Forward Rendering 优缺点

很明显，对于 Forward Rendering，光源数量对计算复杂度影响巨大，所以比较适合户外这种光源较少的场景（一般只有太阳光）。

但是对于多光源，我们使用 Forward Rendering 的效率会极其低下。因为如果在 vertex shader 中计算光照，其复杂度将是$O(num_geometry_vertexes * num_lights)$，而如果在 fragment shader 中计算光照，其复杂度为$O(num_geometry_fragments * num_lights)$ 。可见光源数目和复杂度是成线性增长的。

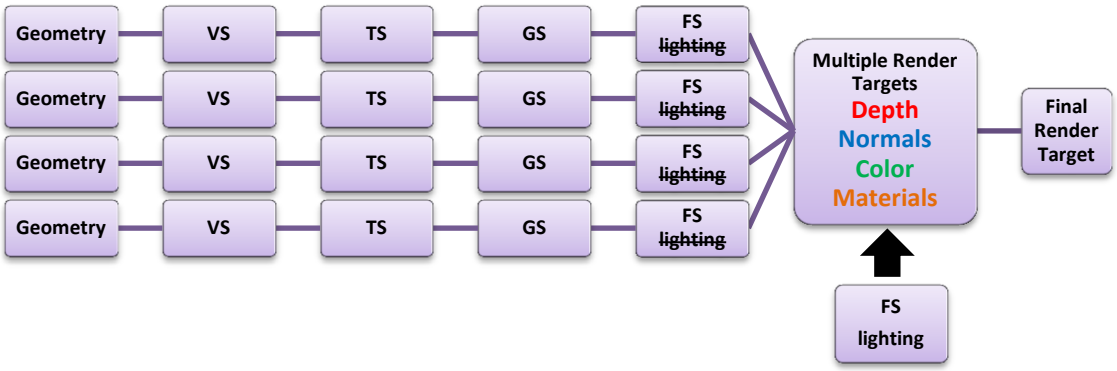
对此，我们需要进行必要的优化。比如

1.多在 vertex shader 中进行光照处理，因为有一个几何体有 10000 个顶点，那么对于 n 个光源，至少要在 vertex shader 中计算 10000n 次。而对于在 fragment shader 中进行处理，这种消耗会更多，因为对于一个普通的 1024x768 屏幕，将近有 8 百万的像素要处理。所以如果顶点数小于像素个数的话，尽量在 vertex shader 中进行光照。

2.如果要在 fragment shader 中处理光照，我们大可不必对每个光源进行计算时，把所有像素都对该光源进行处理一次。因为每个光源都有其自己的作用区域。比如点光源的作用区域是一个球体，而平行光的作用区域就是整个空间了。对于不在此光照作用区域的像素就不进行处理。但是这样做的话，CPU 端的负担将加重。

3.对于某个几何体，光源对其作用的程度是不同，所以有些作用程度特别小的光源可以不进行考虑。典型的例子就是 Unity 中只考虑重要程度最大的 4 个光源。

2.2 Deferred Rendering



Deferred Rendering（延迟渲染）顾名思义，就是将光照处理这一步骤延迟一段时间再处理。具体做法就是将光照放在已经将三维物体生成二维图片之后进行处理。也就是说将物空间的光照处理放到了像空间进行处理。要做到这一步，需要一个重要的辅助工具——G-Buffer。G-Buffer 主要是用来存储每个像素对应的 Position，Normal，Diffuse Color 和其他 Material parameters。根据这些信息，我们就可以在像空间中对每个像素进行光照处理[3]。下面是 Deferred Rendering 的核心伪代码。

For each object:

    Render to multiple targets

For each light:

    Apply light as a 2D postprocess

下面简单举个例子[1]。

首先我们用存储各种信息的纹理图。比如下面这张 Depth Buffer，主要是用来确定该像素距离视点的远近的。

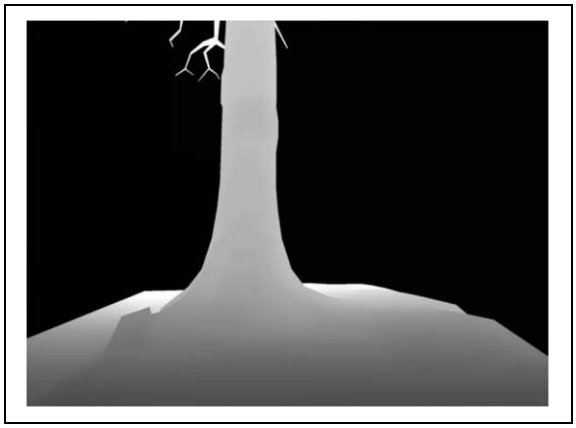


图. Depth Buffer

根据反射光的密度/强度分度图来计算反射效果。

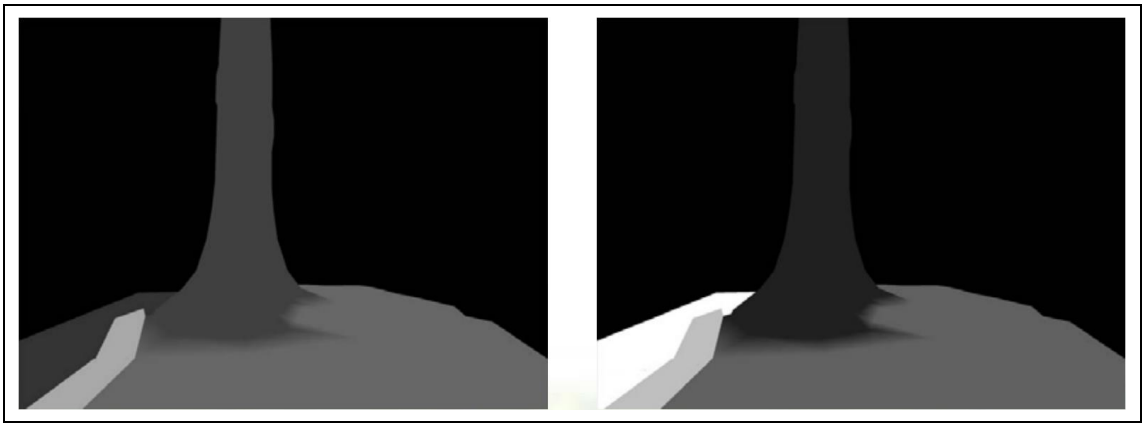


图.Specular Intensity/Power

下图表示法向数据，这个很关键。进行光照计算最重要的一组数据。

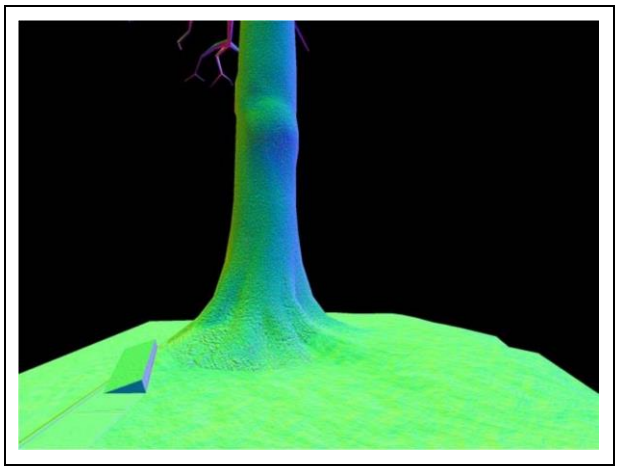


图.Normal Buffer

下图使用了 Diffuse Color Buffer。

图.Diffuse Color Buffer

这是使用 Deferred Rendering 最终的结果。

图.Deferred Lighting Results

Deferred Rendering 的最大的优势就是将光源的数目和场景中物体的数目在复杂度层面上完全分开。也就是说场景中不管是一个三角形还是一百万个三角形，最后的复杂度不会随光源数目变化而产生巨大变化。从上面的伪代码可以看出 deferred rendering 的复杂度为 O(screen_resolution + num_lights)。

但是 Deferred Rendering 局限性也是显而易见。比如我在 G-Buffer 存储以下数据

Depth	R32F
Normal + scattering	A2R10G10B10
Diffuse color + emissive	A8R8G8B8
Other material parameters	A8R8G8B8

这样的话，对于一个普通的 1024x768 的屏幕分辨率。总共得使用 1024x768x128bit=20MB，对于目前的动则上 GB 的显卡内存，可能不算什么。但是使用 G-Buffer 耗费的显存还是很多的。一方面，对于低端显卡，这么大的显卡内存确实很耗费资源。另一方面，如果要渲染更酷的特效，使用的 G-Buffer 大小将增加，并且其增加的幅度也是很可观的。顺带说一句，存取 G-Buffer 耗费的带宽也是一个不可忽视的缺陷。

对于 Deferred Rendering 的优化也是一个很有挑战的问题。下面简单介绍几种降低 Deferred Rendering 存取带宽的方式。最简单也是最容易想到的就是将存取的 G-Buffer 数据结构最小化，这也就衍生除了 light pre-pass 方法。另一种方式是将多个光照组成一组，然后一起处理，这种方法衍生了 Tile-based deferred Rendering。

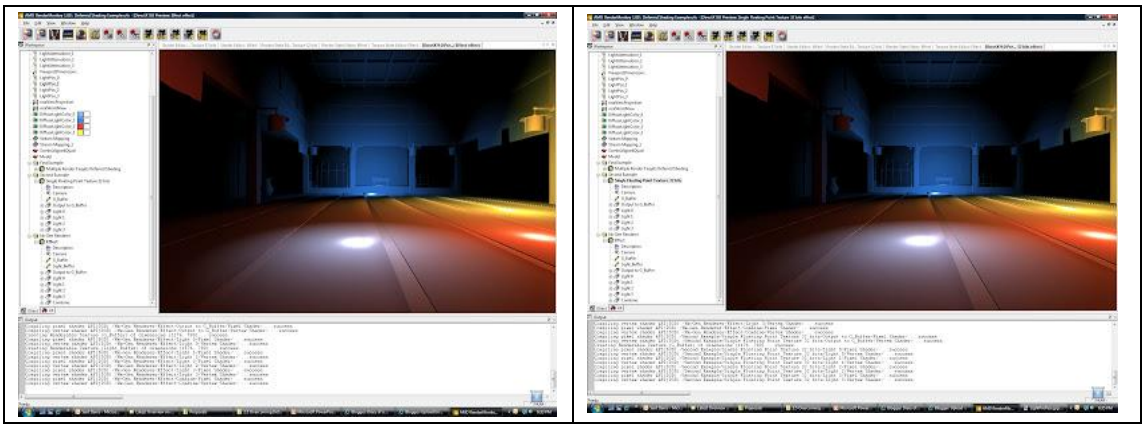
2.2.1 Light Pre-Pass

Light Pre-Pass 最早是由 Wolfgang Engel 在他的博客[2]中提到的。具体的做法是**(1)**只在 G-Buffer 中存储 Z 值和 Normal 值。对比 Deferred Render，少了 Diffuse Color， Specular Color 以及对应位置的材质索引值。**(2)**在 FS 阶段利用上面的 G-Buffer 计算出所必须的 light properties，比如 Normal*LightDir,LightColor,Specular 等 light properties。将这些计算出的光照进行 alpha-blend 并存入 LightBuffer（就是用来存储 light properties 的 buffer）。**(3)**最后将结果送到 forward rendering 渲染方式计算最后的光照效果。

相对于传统的 Deferred Render，使用 Light Pre-Pass 可以对每个不同的几何体使用不同的 shader 进行渲染，所以每个物体的 material properties 将有更多变化。这里我们可以看出对于传统的 Deferred Render，它的第二步（见伪代码）是遍历每个光源，这样就增加了光源设置的灵活性，而 Light Pre-Pass 第三步使用的其实是 forward rendering，所以可以对每个

mesh 设置其材质，这两者是相辅相成的，有利有弊。另一个 Light Pre-Pass 的优点是在使用 MSAA 上很有利。虽然并不是 100%使用上了 MSAA（除非使用 DX10/11 的特性），但是由于使用了 Z 值和 Normal 值，就可以很容易找到边缘，并进行采样。

下面这两张图，左边是使用传统 Deferred Render 绘制的，走遍是使用 Light Pre-Pass 绘制的。这两张图在效果上不应该有太大区别。



2.2.2 Tile-Based Deferred Rendering

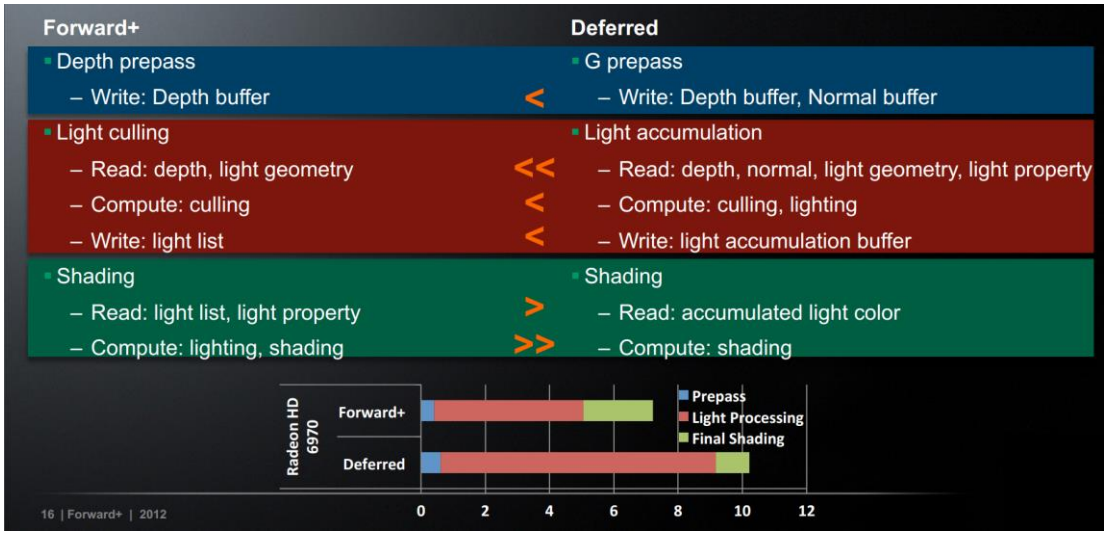
TBDR 主要思想就是将屏幕分成一个个小块 tile。然后根据这些 Depth 求得每个 tile 的 bounding box。对每个 tile 的 bounding box 和 light 进行求交，这样就得到了对该 tile 有作用的 light 的序列。最后根据得到的序列计算所在 tile 的光照效果。[4][5]

对比 Deferred Render，之前是对每个光源求取其作用区域 light volume，然后决定其作用的的 pixel，也就是说每个光源要求取一次。而使用 TBDR，只要遍历每个 pixel，让其所属 tile 与光线求交，来计算作用其上的 light，并利用 G-Buffer 进行 Shading。一方面这样做减少了所需考虑的光源个数，另一方面与传统的 Deferred Rendering 相比，减少了存取的带宽。

2.3 Forward+

Forward+ == Forward + Light Culling[6]。Forward+很类似 Tiled-based Deferred Rendering。其具体做法就是先对输入的场景进行 z-prepass，也就是说关闭写入 color，只向 z-buffer 写入 z 值。注意此步骤是 Forward+必须的，而其他渲染方式是可选的。接下来来的步骤和 TBDR 很类似，都是划分 tiles，并计算 bounding box。只不过 TBDR 是在 G-Buffer 中完成这一步骤的，而 Forward+是根据 Z-Buffer。最后一步其实使用的是 forward 方式，即在 FS 阶段对每个 pixel 根据其所在 tile 的 light 序列计算光照效果。而 TBDR 使用的是基于 G-Buffer 的 deferred rendering。

实际上，forward+比 deferred 运行的更快。我们可以看出由于 Forward+只要写深度缓存就可以，而 Deferred Render 除了深度缓存，还要写入法向缓存。而在 Light Culling 步骤，Forward+只需要计算出哪些 light 对该 tile 有影响即可。而 Deferred Render 还在这一部分把光照处理给做了。而这一部分，Forward+是放在 Shading 阶段做的。所以 Shading 阶段 Forward+耗费更多时间。但是对目前硬件来说，Shading 耗费的时间没有那么多。



Forward+的优势还有很多，其实大多就是传统 Forward Rendering 本身的优势，所以Forward+更像一个集各种 Rendering Path 优势于一体的 Rendering Path。

3. 总结

首先我们列出 Rendering Equation，然后对比 Forward Rendering，Deferred Rendering 和 Forward+ Rendering[6]。

3.1 Rendering Equation

其中点x处有一入射光，其光强为L_e，入射角度为w_i。根据函数f和L_e来计算出射角为w_o处的出射光强度。最后在辅以出射光的相对于视点可见性$V'(w_o)$。注意此处的n为场景中总共有n个光源。

$$\mathrm{L} = \sum_{i}^{n} \{L_e f(x, w_i, w_o) V'(w_o)\}$$

3.2 Forward Renderng

由于 Forward 本身对多光源支持力度不高，所以此处对于每个点x的处理不再考虑所有的n个光源，仅仅考虑少量的或者说经过挑选的m个光源。可以看出这样的光照效果并不完美。另外，每个光线的$V'(w_o)$是计算不了的。

$$L_{\text{forward}} = \sum_{i}^{m} \{L_e f(x, w_i, w_o) V'(w_o)\}$$

3.3 Deferred Rendering

由于 Deferred Rendering 使用了 light culling，所以不用遍历场景中的所有光源，只需遍历经过 light culling 后的$\tilde{n}$个光源即可。并且 Deferred Rendering 将计算 BxDF 的部分单独分出来了。

$$L_{\text{deferred}} = \sum_{i}^{\tilde{n}} \{L_e V'(w_o)\} f(x, w_i)$$

3.4 Forward+ Rendering

可以看出 Forward+和 Forward 最大区别就是光源的挑选上有了很到改进。

$$L_{\text{forward+}} = \sum_{i}^{\tilde{n}} \{L_e f(x, w_i, w_o) V'(w_o)\}$$

参考文献

[1] Shawn Hargreaves. (2004) “Deferred Shading”. [Online] Available: http://hall.org.ua/halls/wizzard/books/articles-cg/DeferredShading.pdf (April 15,2015)
[2] Wolfgang Engel. (March 16, 2008) “Light Pre-Pass Renderer”. [Online] Available: http://diaryofagraphicsprogrammer.blogspot.com/2008/03/light-pre-pass-renderer.html(April 14,2015)
[3] Klint J. Deferred Rendering in Leadwerks Engine[J]. Copyright Leadwerks Corporation, 2008.
[4] 龚敏敏.(April 22, 2012) “Forward 框架的逆袭：解析 Forward+渲染”. [Online] Available: http://www.cnblogs.com/gongminmin/archive/2012/04/22/2464982.html(April 13,2015)
[5] Lauritzen A. Deferred rendering for current and future rendering pipelines[J]. SIGGRAPH Course: Beyond Programmable Shading, 2010: 1-34.
[6] Harada T, McKee J, Yang J C. Forward+: Bringing deferred lighting to the next level[J]. 2012.