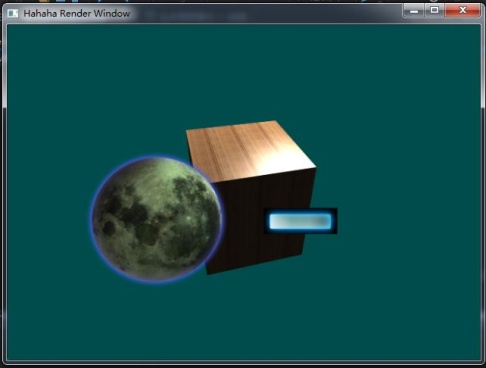
* **NoiseRenderer：**

1. (2015.6.29)负责渲染，暂时最主要是RenderMeshInList，其中RenderList(mesh)储存在FatherScene中。Render时此类会负责把数据按频率分类更新到GPU。并且画图呗。
2. (2015.6.29)设计好的Effect框架和Shader都由此类来管理。
3. (2015.7.7)Renderer负责更新ConstantBuffer。
4. (2015.7.30)Render开始管理Mesh渲染列表。新增画3D线条的功能，拥有一个size为2的VB，用于表示线段的两个顶点。即时画线条，所以每次Draw\_Line3D都会UpdateSubresource一次。专门开辟了一个Cbuffer来优化，只传送V 、P矩阵（妈雕原来我的app和fx端矩阵顺序不一样搞出了奇怪的错误
5. (2015.8.2)开始管理所有渲染列表，线条的储存与更新留给NoiseLineBuffer。fx里面的变量,Cb,texture2D等fx c++借口均在NoiseRenderer的mFunction\_Init里创建，在各种mFunction\_xxx\_update里更新（一般是Draw call之前）。
6. (2015.8.30)各种更低级的几何对象由NoiseGraphicObject来管理
7. (2015.9.16)新增几种blending mode :opaque , additive ,alpha ,multiply，顾名思义。
8. (2015.9.18)在每次渲染（及渲染的数据更新）之前都validate TexMgr/MatMgr/TexID/MatID.
9. (2015.9.24)新增对cube map的支持（如天空盒），更新到gpu之前会检查一下材质ID和纹理类型对不对。
10. (2015.10.10)新增Set SamplerState和DepthStencilState。在drawCall前检查顶点数是否为0以防浪费。
11. (2015.11.10)由于GraphicObject里的长方形可以是文本Texture，所以渲染的时候要先判断一下NOISE\_TEXTURE\_TYPE。
12. (2015.11.12)改为主动式的AddToRenderList，以前是对象主动AddToRenderList，这样还要validate指针好麻烦。（其中Static/Dynamic Text对象是要抽取他们的内部的Graphic Object)

* **FX：**

1. (2015.7.7)。
2. (2015.8.23)现在在光照的时候顺便就用上了贴图。SamplerState只用各向异性的(Anisotropic)，不让用户选了，麻烦。在material里面，如果texture是无效的，那就用原始的纯色diffuse。
3. (2015.9.18)Fog:如果不开雾效就正常计算光照等，开了就在最后根据点到cam的距离进行finalColor 到 fogColor的线性插值。
4. (2015.9.26)Normal Mapping:为了实现法线映射和以后的其他特效，现在LightVec和Vec\_ToCam等向量全部转换到TBN坐标系进行光照。

。

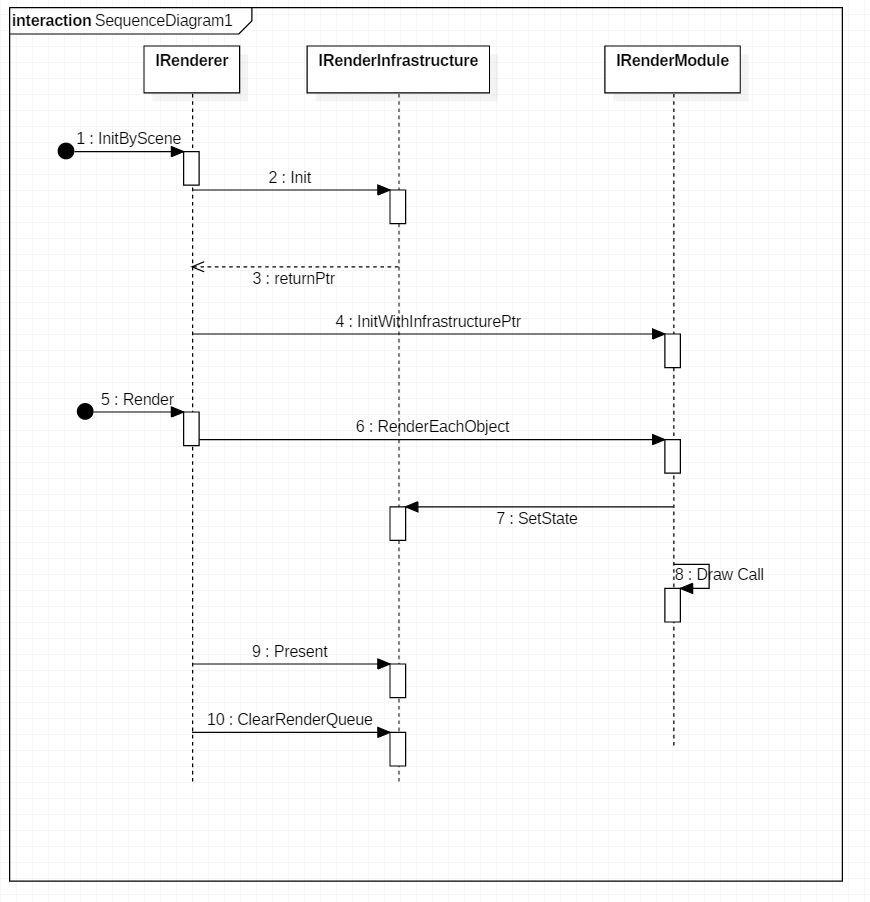
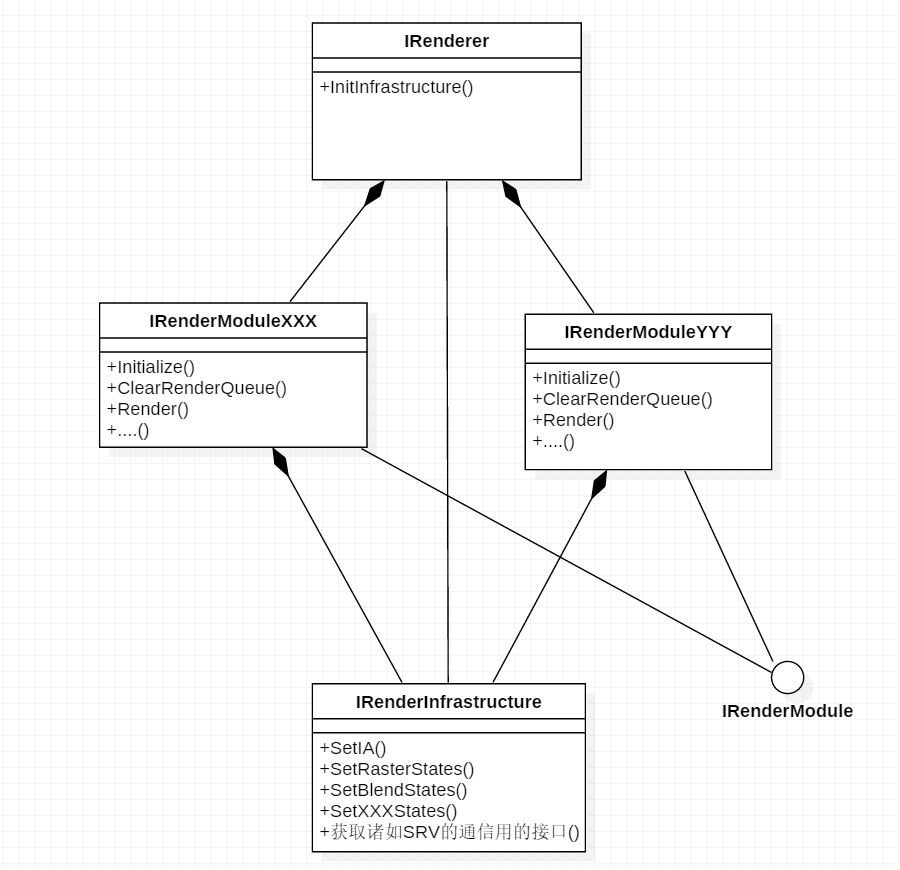
世界X = B上分量对X的贡献+N上分量对X的贡献+T上分量对X的贡献

**Shader : Lighting**

(2015.7.12)暂时是多个光源的三种颜色的叠加：Ambient环境光，Diffuse漫反射，Specular镜面光（高光）。Ambient是程序设定好的作为一个基色。Diffuse是用cos law（Lambertian body），并且有一个mDiffuseIntensity来Scale，这个是view Point independent的颜色，与观察者位置无关，只跟法线和入射光方向有关。Specular是view Point dependent的，与 反射光方向和观察者direction的夹角 有关，并且也有一个mSpecularIntensity来Scale。这个是最基本的模型。三种光有Directional,Point,Spot，实现细节就忽略了，因为差不多。反正就是VS\_OUTPUT需要多一个posW用于得到当前处理点的世界坐标。

**Renderer的模块拆分**

(2018.1.25)每种可渲染的物体是相当于一种渲染数据容器，然后会有对应的RenderModule来实现其渲染准备、提取数据、DrawCall的逻辑。有一些公用的基础设施(infrastructure)例如InputAssembler等的各种D3D pipeline state setting就封装在了IRendererInfrastructure。但可惜Renderer要提取数据的时候不免需要相应的容器类（Mesh、GraphicObject）等要friend提取者，虽然写起来简单是简单了，但是破坏了一定的封装性。每个IRenderModule都要实现一些通用的接口例如Initialize，ClearRenderList，Render等，而每个IRenderModule都会有一份Infrastructure的reference。具体参考下面的UML类图和时序图。



**ShaderVariableManager**

(2018.1.28)原本还用着ConstantBuffer来管理、更新shader variables到gpu，但是一旦我要更改shader和CB的内容，那么我也必须相应地更改c++端的数据结构，还要考虑内存对齐，这怕是麻烦死了。于是就想着学openGL那样，在c++端里面每个variable都绑定一下，然后就可以直接setVarXXXX来更新某个变量数据到GPU了。所以现在就用起了各种Effect Variable系列的接口，所有的Shader Variable都丢给shaderVarManager管理，然后需要用到它的类（例如IRenderer、ICollisionTestor）就获取一个shaderVarMgr的单例(Singleton)指针，调调函数更新数据到GPU，至于初始化变量各种绑定就封装在shaderVarManager里面了。