**整体框架：**

暂时来说，Noise3D库有如下几个类：

NoiseEngine

NoiseCamera

NoiseScene

NoiseAtmosphere(2015.9.18)

NoiseMesh

NoiseRenderer

NoiseLightManager

NoiseLineBuffer(2015.8.2)

NoiseGraphicObject(2015.8.30)

NoiseFileManager(2015.7.22)

NoiseMaterialManager(2015.8.20) //-\_-。欧洲旅游完啦

NoiseUtLayerGenerator(2015.7.25)

NoiseUtTimer

* **General：**

1. (2015.7.7)

更新数据到GPU：在APP中已经定义了跟cbuffer一样的结构体，在app中只需更新结构体，并在NoiseRenderer::m\_Function\_UpdateCbXX中赋值，注意要和fx中的结构对齐，也要注意内存对齐（4D 向量）。需要更新的文件有(NoiseType.h,NoiseRenderer.cpp,fx)

* **NoiseEngine：**

1. (2015.6.29)这是作为最一开始用来初始化的东西（其实还没它什么事）。或许以后可以把Win32窗体也的创建也整合进去。

* **NoiseCamera：**

1. (2015.6.29)顾名思义，用来模拟一个灵活的摄像机，基本参数有位置，姿态角，视场角，Near/Far Plane，Aspect Ratio。还顺带管理与它自己相关的View Matrix和Projection Matrix。
2. (2015.6.29)NoiseCamera所用的姿态角是Tait-Bryan angle，即Yaw-Pitch-Roll(Y-X-Z)系统
3. (2015.6.29)此类管理的两个矩阵不会在每次更新Position和Rotation的时候更新，是在渲染的时候由NoiseRenderer来更新（友元类）。但是，这个类一个比较好的设计是，更新Position或Lookat的时候可能会带来姿态角的变化，所以会更新姿态角；同理更新Rotation的时候，也会更新Lookat和Direction。
4. (2015.6.29)由于Proj Matrix不需要经常更新，要修改时才需要修改，所以设置了一个Dirty标记(目前叫mCanUpdateProjMatrix)用于判断是否需要更新到GPU。
5. (2015.6.29)未来需要设计一些Special Movement，如绕轴转，渐变到某姿态/位置（各种插值方式），还有景深，聚焦等。

* **NoiseScene：**

1. (2015.6.29)以后这个就是管理3D场景渲染的主心骨啦，跟Mesh,Renderer,Camera等都互粉了（双向friend）。所以它是管理着各种对象列表，m\_pChildMesh, m\_pChildCamera, m\_pChildRenderer （等）
2. (2015.6.29)Scene类暂时还管理着RenderList(Mesh)，这是为了强化它“对象列表管理者”的地位。
3. (2015.6.29)所有的场景对象都要由它来Create。
4. (2015.8.2)渲染列表交给Renderer来管理。

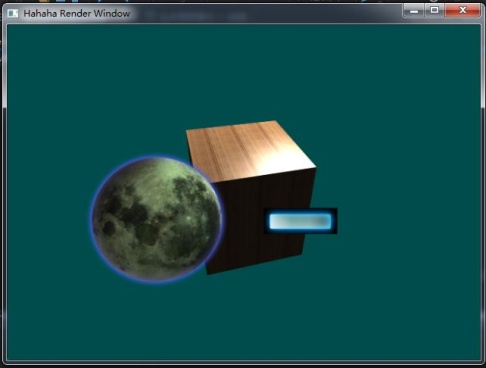
* **NoiseAtmosphere：**

1. (2015.9.18)Fog : 需要参数near和far和color，之后在pixel shader里根据distance from current point to camera来进行颜色的线性插值(在PS的最后进行插值)。关于开启fog特效有两个BOOL变量，一个是Enabled,另一个是在整个Atmosphere AddToRenderList() 之后才开启(渲染后才关闭)，设置两个BOOL是为了实现当Atmosphere添加到渲染列表且有效之后才开启，因为fog在shader里是集成到原有的计算中，并没有独立出一个tech，所以这个为了实现所有隶属于Atmosphere的东西能在AddToRenderList之后才能画出来。（不然如果只有一个Enabled，就算atmosphere没有被addToRenderList，fog也可能被画出来）
2. (2015.9.18)SkyDome（穹顶）：类内内建VB/IB，用N\_SimpleVertex（因为不需要法线）。不采用NoiseMesh做天空实体原因是sky dome有可能是扁的，不想只能创建球体，所以就copy的createsphere改改来创建sky dome。

* **NoiseMesh：**

1. (2015.6.29)这是最基本的模型类了，所有Vertex和Index创建后都不需要改变了。
2. (2015.6.29)它自己也管理着一些属性，如Position,Rotation。当然顺带管理的就是World Matrix。它有自己的VB/IB，这样的结构比较清晰，当然以后要进行优化，因为有太多零散的小VB时，在执行Draw Call的时候代价较大，GPU需要等待CPU的传递命令时间。
3. (2015.6.29)Mesh要渲染的时候可以执行Add To Render List。
4. (2015.6.29)Mesh自带几种创建几何体方法。下面会提到具体。
5. (2015.7.7)现在会在类内管理一个Material结构体。
6. (2015.7.22)继承了NoiseFileManager，用于加载文件
7. (2015.8.20)准备实现多纹理(texture)和多材质(material)，这样的话要引入Mesh Subset(子集)的概念。我的定义是：具有同样纹理和材质的图元（三角形）集合，每个子集在shader里用同一个设置来Draw。因为Texture和Material都是用int来确定的，所以我们要先对三角形列表的TexID和MatID属性进行排序（应该要用到<algorithm>），然后再搞一个列表记录subset的区间端点（都是UINT）。
8. (2015.8.20)如果Material/Texture 的 ID不合法，会用NOISE\_MACRO\_INVALID\_XXX\_ID来表示，值为-1。
9. (2015.8.22)每个mesh必须有material。在mesh初始化时设一次user-set 的material（设为默认），用户也可以自己手动设置mesh的material ID（有一个专门的UINT来储存，就是防止用户在CreateXXX之前就设了material），在CreateXXX里面也会设置这种user-set 的 material。每次给mesh设定material都会更新mesh的图元信息列表和子集列表（subset）
10. (2015.8.22)没有无效的Material，如果设置了无效的mat ID就要把它的mat ID设为0（default）。（贴图可以不要，但是材质必须要有）

* **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
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.

* **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。

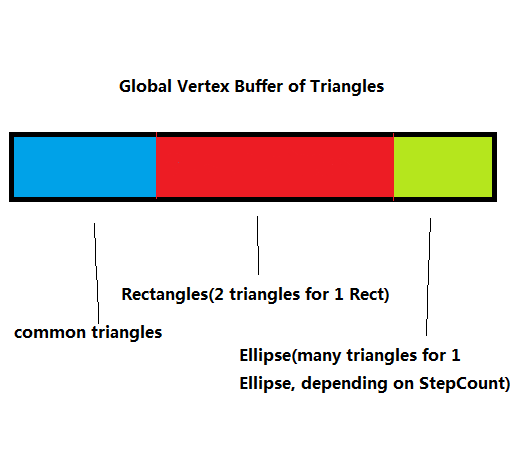
* **NoiseLightManager：**

1. (2015.6.29)负责管理场景的灯光，有三种光源Directional,Point,Spot。每种光源都有相同的属性如Ambient , Diffuse , Specular，但Point和Spot会有更多参数。
2. (2015.6.29)之所以material也会有与light类似的参数，是因为最后在shading的时候需要有灯光与材质的交互。
3. (2015.7.12)把灯光分为了静态光源和动态光源，并且都设置了最大数目。这样分开是做一定的优化，减少静态光源不必要的更新。然后增加了光源各种ADD,REMOVE。新增动态光源传入的参数是灯光的指针，所以可以在app中改变灯光的struct便可以更新灯光状态。

* **NoiseFileManager：**

1. (2015.7.22)纯粹加载文件到vector<char>；导出文件(vector<char>)
2. (2015.7.22加载stl文件，要求输入vertex\index\normal\info 的buffer指针。
3. (2015.8.2)新增导入/导出 NOISELAYER切层文件，自己瞎编的一个格式，详情参阅下一部分的 2015.7.30 NoiseUtSlicer

* **NoiseLineBuffer：**
* **NoiseGraphicObject**

1. (2015.8.2)这是一个专门存线条的buffer，本来是打算直接在renderer那存个两顶点的vertex buffer，每次调用render 的时候就更新数据到GPU。 有AddToRenderList，在这个函数里会检查是否需要更新到GPU缓冲区，并调整GPU缓冲区的大小。
2. (2015.8.30)现在简单基本的几何对象都用这个类来储存（避免即时更新到GPU的速度下降）。暂时有几种类型，Line3D/2D，Point3D/2D，Triangle2D，每种类型都有自己的状态变量和Vertex Buffers。
3. (2015.8.30)Triangle2D可以用来装圆，粗线条，切角长方形，正方形，贴图长方形...等复杂一点点的的对象，这就要开辟多一些对象信息列表来给Triangle2d的Vertex buffer分区。
4. (2015.9.5)现在有了实心矩形和椭圆的渲染（还没贴上图）。有一些特殊的列表，专门记录line / triangle buffer特殊的部分，因为例如长方形或者椭圆这种由一堆三角形组成的图形，需要知道对应的三角形id段，暂时用startID和elementCount来表示。普通三角形的ID会出现在一个vector<UINT>里面。
5. (2015.9.5)在AddToRenderList的时候，需要更新才进行updatesubresouce()；
6. (2015.9.6)现在有了矩形和椭圆线框。要注意Line2D和Triangle2D的vertex buffer比较特殊，有一定的分区，如下图。
7. (2015.9.16)Vertices的更新都在AddToRenderList()里面

* **NoiseMaterialManager：**

1. (2015.8.20) 管理材质，有名字或者Index两种访问方式，名字用std::string。最主要使用UINT类型的Index来访问纹理，NoiseMesh里面最终也是设置通过设定ID来设置纹理。若某个三角形没有设置纹理，则在shader里面的gIsXXXMapValid会设为0，关闭此三角形Texture mapping。
2. (2015.8.22)Mat Mgr的第一个元素（下标是0）是在构造函数中就创建的默认材质。
3. (2015.8.22)Material包含了基本颜色和贴图ID
4. (2015.8.22)没有无效的Material，如果哪个类设置了无效的mat ID就要把它的mat ID设为0（default）。（贴图可以不要，但是材质必须要有）。无效贴图的tex ID为-1。

* **NoiseTextureManager：**

1. (2015.8.22)管理纹理。无效贴图的tex ID为-1。

**部分技术实现：**

**————2015.6.27————**

**NoiseCamera::mFunction\_UpdateRotation**

这个是更新Lookat或Position的时候触发的，更新姿态角。翻滚角不需要更新。

****

大致按着这个图，搞搞反三角函数就可以了。不过就是注意如果Direction是(0,y,0)，即看着正上方时，是不可以用atan2()的，这时要讨论。

**————2015.6.27————**

**NoiseCamera::mFunction\_UpdateDirection**

这个是更新姿态角的时候触发的，用于更新Direction和Lookat。

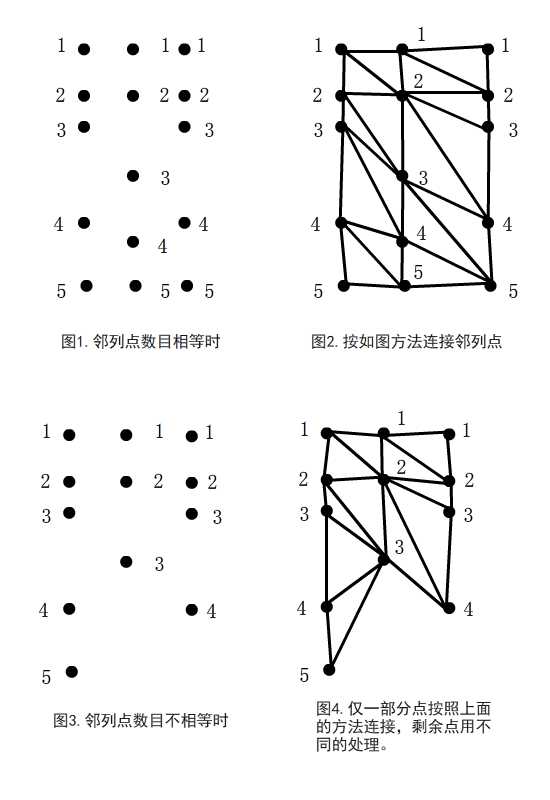
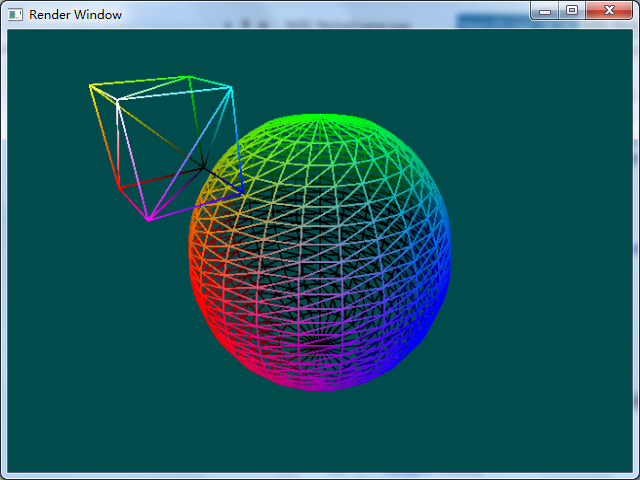
****

还是大致按着这个图，搞搞三角函数就可以了。

**————2015.6.29————**

**NoiseMesh::CreateSphere**

(2015.6.29)用的基本思路是当年做的3D扫描仪时圆柱体模型，用一张包着球体大网来生成球体网格。

(2015.6.29)如上左图解释，图1画出了3列点。当我们只考察第一与第二列点时，设函数L1(x)的返回值为第一列的第x个点，L2(x)的返回值为第二列的第x个点，T(p1,p2,p3)得到一个以p1,p2,p3为顶点的三角形，则假设每列点都有N个点，则设三角形集：

三角形的集合**E**为，

（呵呵论文上复制下来的，我的天这还是高二写的）

(2015.6.29)所以其实通俗的讲就是根据iRingCount的数量来确定横切片的数量，用iColumnCount来确定纵切片的数量，先生成侧面的网格。注意由于以后要加上纹理坐标，故第一列要复制多一份（因为接缝处纹理坐标不一样）。还要注意iColumnCount是不算重复的那一列的。最后再单独讨论TOP/BOTTOM的封顶问题（TOP顶点和BOTTOM是单独生成的）。反正就是这样了，代码那里可以慢慢纠结，这里给个思路。

(2015.7.16)生成顶点的时候，会有当前处理的列ID和环ID，把这两个ID分别映射到【0，1】，生成纹理坐标，就当是纹理的球形包裹方法。

(2015.8.23)生成y坐标时的步数(step)从y坐标换成了空间角，生成更均匀分布的顶点。

**————2015.6.29————**

**NoiseMesh::CreateCylinder**

(2015.6.29)其实跟CreateSphere没多大区别....Index都是同一种方法生成的的，只是多了个参数Height，Generate Vertex和其他稍作修改便可。

(2015.7.5)在顶点属性增加了Normal之后，一切都变得坑爹。然后就增加了TOP/BOTTOM两个重复的Ring，用于生成不同的法线。

(2015.7.16)这个柱形纹理坐标的生成会有点特别吧，texCoordX还是照旧和球体类似，但是texCoordY是把 ( (半径x2) + 高)的这段长度映射到【0，1】，以确保两个顶面和侧面都不会有太大的纹理拉伸。

**————2015.7.12————**

**Shader : Lighting**

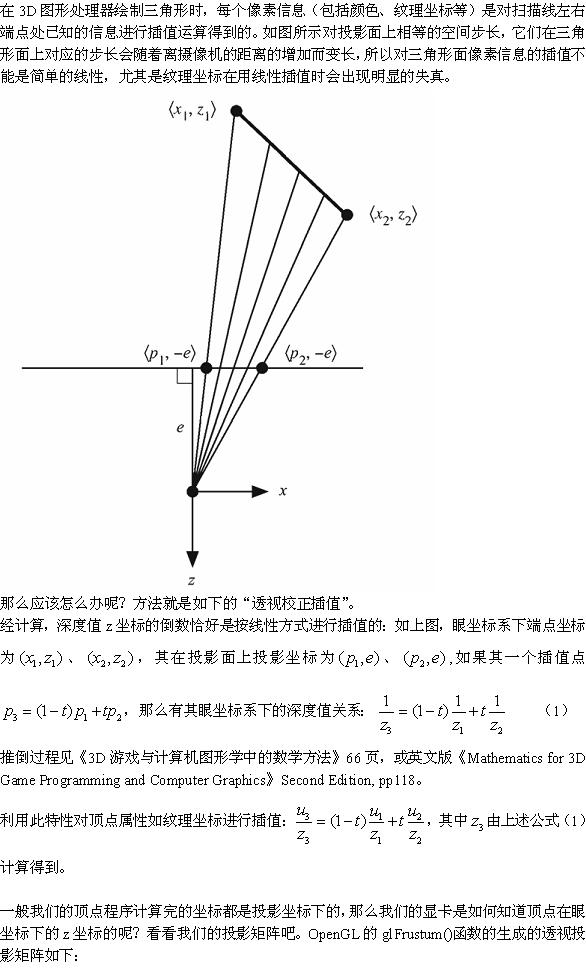
(2015.7.12)暂时是多个光源的三种颜色的叠加：Ambient环境光，Diffuse漫反射，Specular镜面光（高光）。Ambient是程序设定好的作为一个基色。Diffuse是用cos law（Lambertian body），并且有一个mDiffuseIntensity来Scale，这个是view Point independent的颜色，与观察者位置无关，只跟法线和入射光方向有关。Specular是view Point dependent的，与 反射光方向和观察者direction的夹角 有关，并且也有一个mSpecularIntensity来Scale。这个是最基本的模型。三种光有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。

**————2015.7.15————**

**透视校正插值**

这个烂东西直接在3D空间中推是非常蛋疼的，一开始还想在屏幕空间的插值点那引出一条Screen Ray，然后交空间三角形交于某点，然后消参之类的，求出Screen Space和3D Space插值坐标的关系。但对于这种插值，各个分量是可以分开进行的，所以我们不妨从侧视图或者俯视图下手，探讨2D平面上的“透视校正插值”，然后再推广到3D空间。那其实平面几何会好解决很多，推是没问题的，但是没有比较直观清晰的feel（特别是那个畏畏缩缩的深度倒数是线性插值，简直吃雕）。那= =不用理我就先不推了（？）

还是比较希望能有清晰的感觉。



**————2015.7.22————**

**加载STL**

唔，想不到在VB里面实现很容易的东西在c++还有那么多手尾。加载主要还是用ifstream，然而是一个一个char的读。之后reinterpret的时候又出问题了，把4个char合成一个int或者float的时候，本来想着是用位移再合并。然后在类型转换的时候还是出了问题，static\_cast之类的会做处理，不是单纯的复制二进制，所以4char合1float的时候还是用到了memcpy。

然后还有一个坑爹的地方，float或者int那种东西的储存大部分机器都是低位在前，跟人类思维有点不同...坑死了.....结果移位是把最后一个char移到最前。

int tmpInt = static\_cast<int>(c1) | static\_cast<int>(c2) << 8 | static\_cast<int>(c3) << 16 | static\_cast<int>(c4) << 24;

STL的构造：（百度百科）

二进制STL文件用固定的字节数来给出三角面片的几何信息。

文件起始的80个字节是文件头，用于存贮文件名；

紧接着用 4 个字节的整数来描述模型的三角面片个数，

后面逐个给出每个三角面片的几何信息。每个三角面片占用固定的50个字节，依次是 :

3个4字节浮点数(角面片的法矢量)

3个4字节浮点数(1顶点的坐标)

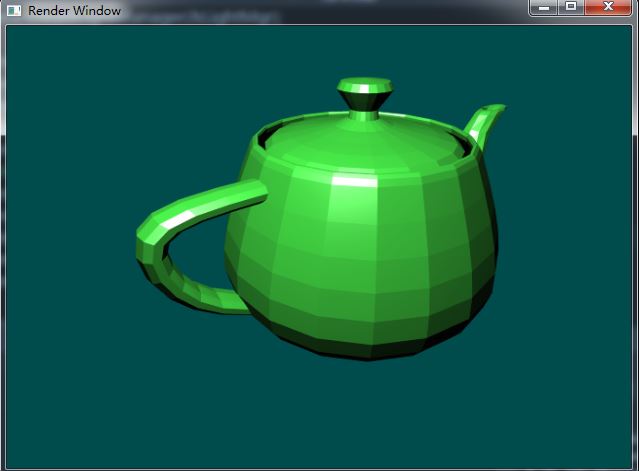
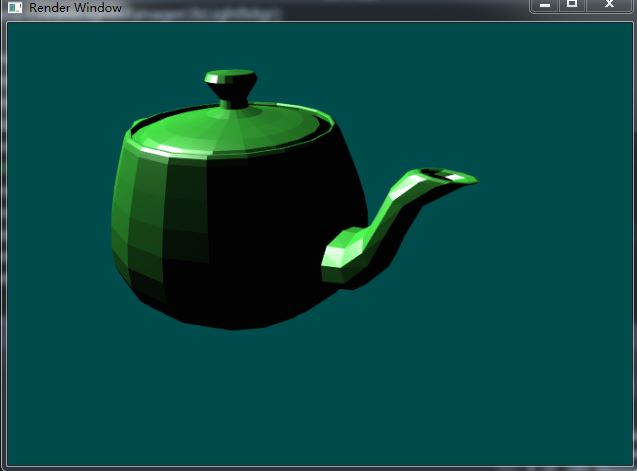
3个4字节浮点数(2顶点的坐标)

3个4字节浮点数(3顶点的坐标)

三角面片的最后2个字节用来描述三角面片的属性信息。

一个完整二进制STL文件的大小为三角形面片数乘以 50再加上84个字节。

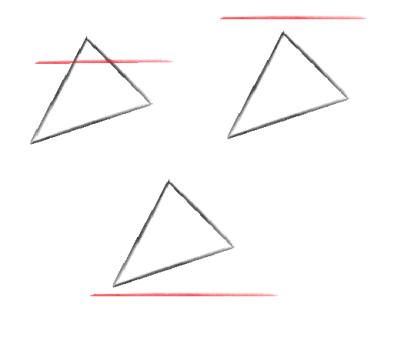
**最后，上图**



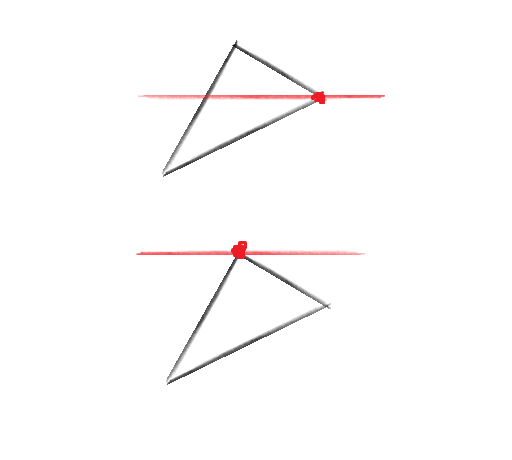
**\*\*\*\*\*\*\*\*\*————2015.7.30————\*\*\*\*\*\*\*\*\*\*\***

**NoiseUtSlicer**

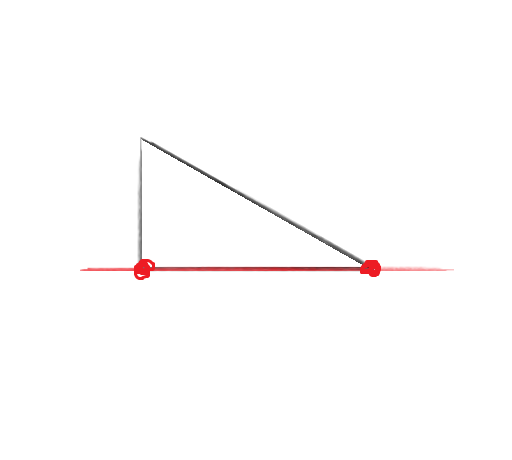
**3D打印机切层器**

1. (2015.7.30)加载STL
2. (2015.7.30)开始切层，每个水平面(y=??的平面)和每个三角形进行切割，一开始先按有多少个三角形顶点在当前切割面上进行分类讨论（因为生成线段的方式有点不一样）。一共有4类，就是有0、1、2、3个顶点在切割面上。反正经过判断切得顶点的个数之后，切得两个顶点的，就能生成一条单独的线段，加入Line\_Segment的buffer之后再处理。最特殊的，三角形和切割面共面，此时切得点有3个，这个三角形直接加入轮廓Buffer（想办法判断哪一边是实心的）（用LineStrip /轮廓 表示的区域）。
3. 

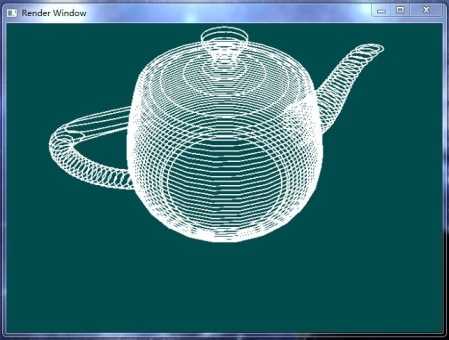
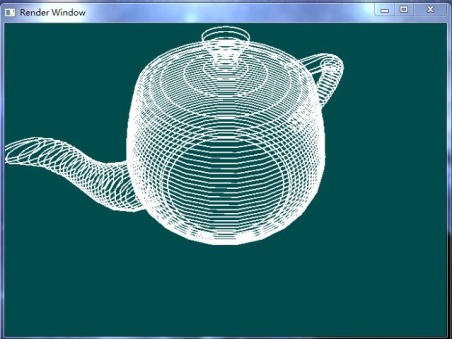
0个顶点在切层上，只能尝试用边去切割

1. 

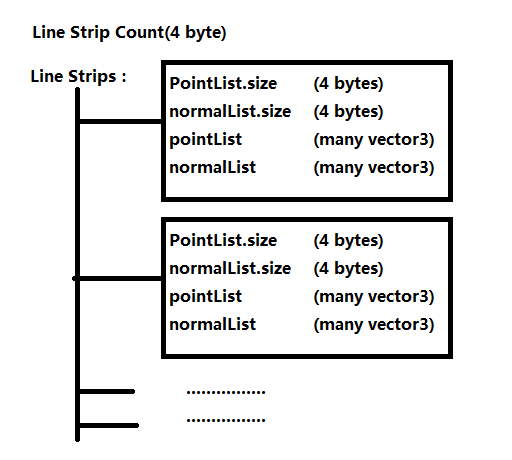
1个顶点在切层上，看看可不可能切出第二个顶点

1. 

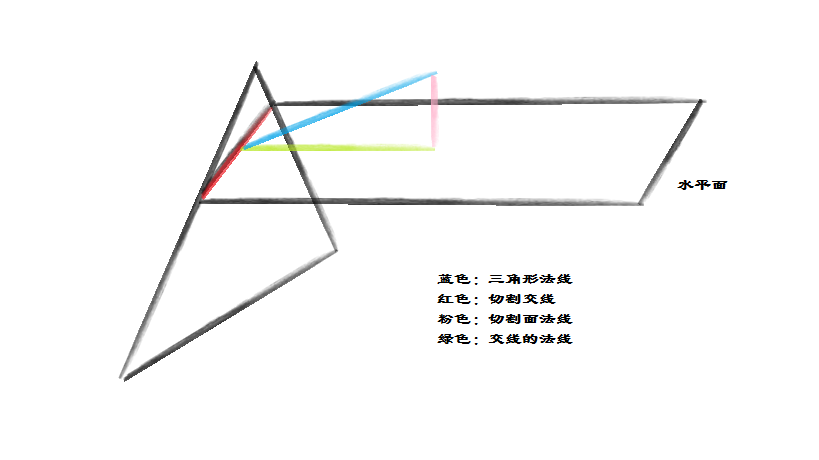
两个顶点在切割面上，直接就是想要的线段。

1. (2015.7.30)把Line\_Segment连起来，连成LineStrip。有个问题，可能到某个顶点时会有多个分支，可能要用到树（？）。再说再说
2. (2015.8.1)不如设定每条Line Strip要首尾相等（共点）才是有效闭合折线
3. 
4. (2015.8.2)好吧，如果STL文件的网格没有缺陷（完全封闭的话），那个Vertex On Layer = 3 的情况都简直可以去掉了，因为肯定有Line Segment会和那个三角形的边重合，这样搞下去的line strip会有重合的部分，麻烦死。而且，Vertex On Layer =3 意味着三角形法线竖直向上，这样的话，边法线也不好确定。所以Intersection的Case 3去掉。（尼玛我在讲什么= =）
5. (2015.8.2)因为LineStrip的顶点是一连串的，所以现在规定线段NormalList的元素个数要比PointList少一个，第几个normal对应第几条线段
6. (2015.8.3)导出NOISELAYER文件：自定义切层格式。全文件第一个四字节是记录Line Strip的个数4 byte的"magic number"——把'k''A''s''T'塞到一个UINT32里面，然后4byte Version ID 暂时是0xffffff01，然后4 byte Line Strip个数。

然后之后是一段一段的Line Strip数据。对于每一段line strip，4 byte的int的层ID (layer ID)，然后4 byte int是PointList的point个数，再一个4 byte int是normalList的normal个数，是比point少一个，因为n个连续点有n-1条线段，也就是n-1条线段法线。之后就是PointList的12 byte 的vector3；之后再normalList的12 byte 的Vector3。（vector3是三个float,xyz分量）



（这个只是示意图，文件的区块逻辑大概是这样，但是新版本加了magicNumer和版本号和layerID）

1. 

(2015.8.2)推论：交线段法线是三角形法线在切割面上的投影。

证明：

∵蓝线⊥三角形

∴蓝线⊥红线

∵粉线⊥水平面

又∵红线水平面

∴ 粉线⊥红线

∵

∴红线⊥面<蓝绿粉>

∴红线⊥绿线

绿线即线段法线，所以三角形法线在切割面上的投影就是线段法线