**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)

NoiseFont套装(2015.11.9)

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

NoiseUtLayerGenerator(2015.7.25)

NoiseGUI套装(2015.10.23)

NoiseUtInputEngine(2015.10.1)

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。
2. (2015.9.x)封装了一个主循环，创建win32窗口的函数，方面快速创建渲染程序。有ReleaseAll函数。

* **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。

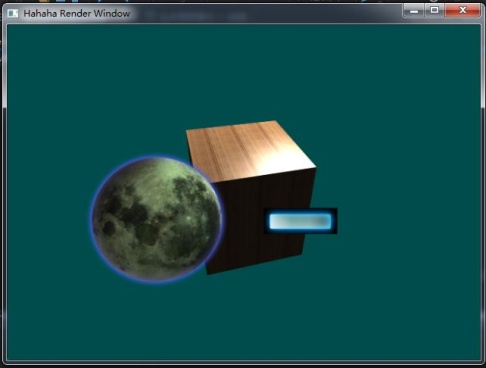
* **NoiseAtmosphere：**

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。
3. (2015.9.24)新增SkyBox。材质必须用CubeMap，建议cubemap为正方形，且边长是2的n次方（？？？）。在更新材质之前会检查tex ID的纹理类型（SRV description的dimension会是d3d11\_srv\_dimension\_cube之类的）。在绘制skybox的时候会把skybox的三边长度上传到gpu，用来把skybox映射到单位正方体上。
4. (2015.10.10)日了狗，用vs2015的图形调试截屏的时候，截屏函数貌似用到了d3dx11Effect，我的skydome的texture尺寸稍微大了点(4096x4096)，就给我爆炸了= =这bug坑的飞起。

* **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。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

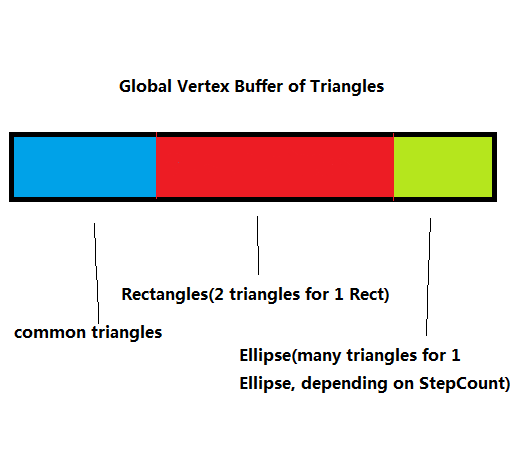
* **NoiseLightManager：**

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

* **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。
8. (2015.9.16)普通(Common) Triangles是不带贴图的，长方形和椭圆可以带贴图。
9. (2015.11.10)作为文本材质的载体，Rectangle可以设置纹理坐标（专门为动态文本的渲染而设置）

* **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。然后在N\_Material里面有一个结构是N\_Material\_Basic，这个是一些基本数值的设置，然后N\_Material里面各种贴图ID就顾名思义了，可以用Noise Texture Manager的GetIndexByName来获取Tex 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。
2. (2015.9.24)内置Validate Texture ID的函数，可以检查ID是否越界，Shader resource view是否为对应的类型(common / cubemap/volumn)。
3. (2015.9.24)新增对CubeMap的支持。（暂时只能从设置好的dds中加载）
4. (2015.10.30)现在TextureObject会多一个vector用于在内存储存图片，我们叫这个是**内存副本**，在创建Texture的时候有一个BOOL可以选择是否创建内存副本。如果不用创建，直接D3DX11CreateSRV创建usage\_default的SRV的就好。但是要留一个内存副本的话，暂时的解决方案就是首先用D3DX11CreateTextureFromFIle创建一个Usage\_Staging的texture，然后Map()回来copy进vector里，之后用这个内存副本初始化一个usage\_default的texture并创建SRV。这里有个坑，用d3dx加载图片map回来之后RowPitch(Width)会有所变化，大概是有点取整的感觉，例如638px的宽会被拉伸成640,1238px被拉成1240px.
5. (2015.10.30)可以把有内存副本的Texture转换为灰度图和法线贴图。灰度图可以调rgb权重（还没归一化），法线图可以调高度场的起伏程度。
6. (2015.10.31)可以修改纹理的内存副本并更新到GPU。出于性能考虑建议先用UINT储存材质ID，不要在循环里用GetIndex （还有GetIndex以后用std::Map来实现Index-Name的配对）
7. (2015.11.10)由于字体纹理/纹理查询表被fontMgr委托由TexMgr管理，这些纹理不能被随便修改和删除，所以纹理应该有访问权限(ACCESS\_PERMISSION)，由FontMgr创建的纹理就应该由FontMgr来裁决而不能被用户操作影响，所以每个TextureObject都会有ACCESS\_PERMISSION属性，每次validateTextureID时如果PERMISSION不吻合就会返回INVALID\_TEXTURE\_ID。
8. (2015.11.18)GetTextureID的线性查找真是慢的吃屎，没什么事不要在循环里用GetTextureID。

* **NoiseUtInputEngine：**

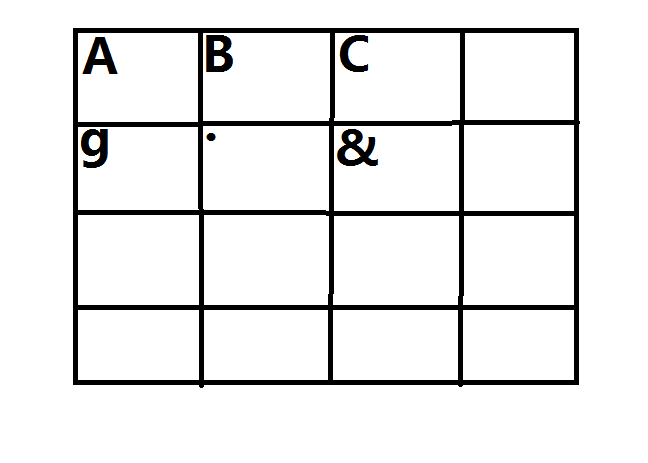
1. (2015.10.1)用了DInput，一开始要Initialize，就要SetCooperativeLevel，然后要Acquire设备才能GetState。有一个Update()函数是用来Validate和Update Mouse/Keyboard 的state，之后就可以用IsKeyPressed等函数来从外部检查按键状态。暂时只支持traditional Mouse/Keyboard，每个按键的state要 &0x80 （doc说的），因为按键state的dwFlag只用到了一半的位。鼠标的State格式用默认的DIMOUSESTATE。
2. (2015.10.1)因为有个接口是获取鼠标的绝对坐标，万一鼠标lost device的话坐标来不及更新，那就要在下一次更新的时候用windows API的GetCursorPos来更新绝对坐标了。
3. (2015.10.2)在Initialize的时候会尝试Acquire设备，但是为了防止init的时候因为焦点不在窗口而失败，init时acquire失败不当成失败。然后每次update的时候都会尝试Acquire，acquire失败就不更新设备信息。

* **NoiseGUI:**

1. (2015.10.20)NoiseGUI包括一个NoiseGUIBasicContainerInfo的基类，NoiseGUIMananger，NoiseGUIText,NoiseGUIScrollBar
2. (2015.10.20)NoiseGUI用了**基于事件**的机制，由NoiseGUIManager来DispatchMessage。需要**传递一个InputEngine的指针**拿来用，判断鼠标状态等。还需要传入渲染窗口的HWND来获取各种参数。
3. (2015.10.20)NoiseGUIBasicContainerInfo基类概括了大部分控件会用到的参数，例如Texture,Position,Size（但是不知道为什么在exe中派生类的基类函数对Intelli Sense不可见导致要全部函数都要对应的定义一个接口才不画红色波浪线，虽然编译是正常的，好鬼烦）。
4. (2015.10.20)NoiseGUIManager里面内置了GraphicObject，AddToRenderList实际上也是把这个graphicObj加到renderList。(2015.11.12)改成每个子对象都内置一个GraphicObj，从而让用户更好地选择设置的可见性等属性
5. (2015.10.23)NoiseGUIManager，用于创建、销毁控件等，还有用于接受鼠标等输入信息来判断控件状态并**发送消息**。（用子控件储存的函数指针）
6. (2015.10.23)NoiseGUIButton：如果exe有需要进行事件的处理，还需要传入**回调函数**指针，有点像windows消息机制，消息处理的回调函数的输入参数是NOISE\_GUI\_EVENTS，用户在exe里可以用switch来处理对应事件。(2015.11.23)消息的获取可以是主动式(GetEventList)或者是被动式(Callback Function)。**callback function由manager来call**，控件只负责储存函数指针。（以前用的VB甚至一个事件对应一个消息处理函数，现在设计是一个控件一个函数）。所以在exe中，在对应的消息处理函数中需要用switch或if等来判断传入的NoiseGUIEvents的值来做相应的处理。
7. (2015.10.30)NoiseGUIButton： 现在Button可以通过SetDragableX（或者Y）来决定是否可以鼠标拖拽（按下左键时）。现在为了简化起见拖拽的时候按键中心会跟着鼠标(2015.11.26)会记录下第一次按下左键时button中心和鼠标的坐标的偏移，这样拖拽就不会让button突然跳跃（用鼠标相对位移叠加的话会因为有延迟导致效果不太好）。因为窗口大小可能会有变化，所以会有一个scaleFactor，用于适应这种缩放，把屏幕空间的鼠标坐标转换到BackBufferSpace的坐标，这样GUI的鼠标判断才不会有偏移。

* **NoiseFontManager:**

1. (2015.11.10)因为字体和文字是个很麻烦的东西，所以引入一个fontMgr来专门管理字体的文字。首先核心是外部库FreeType 2.6.1，这个ANSI C写的外部库可以比较方便的获取字体的位图（灰度图），然后再由我来使用。FT在Set\_Pixel\_Size之后确定了字体最大边界大小，然后可以RenderToGlyph来把字体画到一个缓冲区中（缓冲区大小可能会有变化，不同的char/Wchar 大小会不同，中文比较大，Ascii比较小，而且他们之间的大小也有不同）。
2. (2015.11.10)先实现GetBitmapOfChar（bitmap大小可能变化），然后用一定的排版实现GetBitmapOfString（用户确定Width，函数来排版让每个Char完整显示，但是超过Height的部分会被截断）。
3. (2015.11.10)由于GetBitmapOfString真是慢出屎来，所以动态文本暂时只打算支持Ascii，不然每次更新文本都要调用GetBitmap就真是卡死了。静态文本直接就是一个Bitmap Texture；动态文本可以随时修改。在创建某一个字体时，fontMgr会在TextureMgr里面创建一个Ascii Bitmap Table，这个Texture装了Ascii码为0~127的char的bitmap，暂定是8行16列(16x8)，然后每个char用GraphicObject里面的Rectangle来渲染，修改文字的时候只需要修改Rectangle的纹理坐标，这样效率会高很多。在创建这个AsciiBitmapTable时，由于每个char有图像部分的实际Bounding box尺寸可能比字体尺寸要小，所以这个bitmap Table储存了每个Ascii Char的实际图像大小（fontObject&作为参数之一传到mFunction\_GetBitmapOfChar里，charSizeList会被写入尺寸列表）。如图，因为FreeType得到的像素矩阵都是字符最小尺寸(渲染出一个bounding box)，然后我又把这些小图片放到每个char对应格子的左上角，所以图片定位要小心一点。（每个Rectangle都设为字体的大小，而不是char的实际大小）。然后由于这种坑爹属性，还要在Alignment上花点功夫，太坑了



1. (2015.11.10)用FontMgr来创建StaticText或者DynamicTextA，这些装载着Text的子类内置GraphicObj。而且这些GraphicObj
2. (2015.11.19)直接在FontMgr里面内置一个TextureMananger，不要跟用户创建的材质混淆，所以StaticText的文本纹理和DynamicText的字符位图查找表都创建在fontMgr内部的TexMgr。
3. (2015.11.19)Dynamic Text每次更新都要先看看文本char个数和graphicObject里rect的个数，然后调整rect的个数来适应char个数，之后每次都更新纹理坐标（在字符位图查找表的哪一小块位图，即一个char）。还要更新rect的位置，这个要根据具体的字符来调整上下对齐的y坐标偏移。每个rect以左上角作为“光标”，每设置一个rect就移动光标，同一行光标往右跳，到行尾时光标适时跳到下一行。**超出y边界的字符暂时不去掉。**
4. (2015.11.20)有一个全局函数用于生成字符纵向对齐偏移(Y offset)，可以供GetStringBitmap和Dynamic text的对齐使用。

**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。

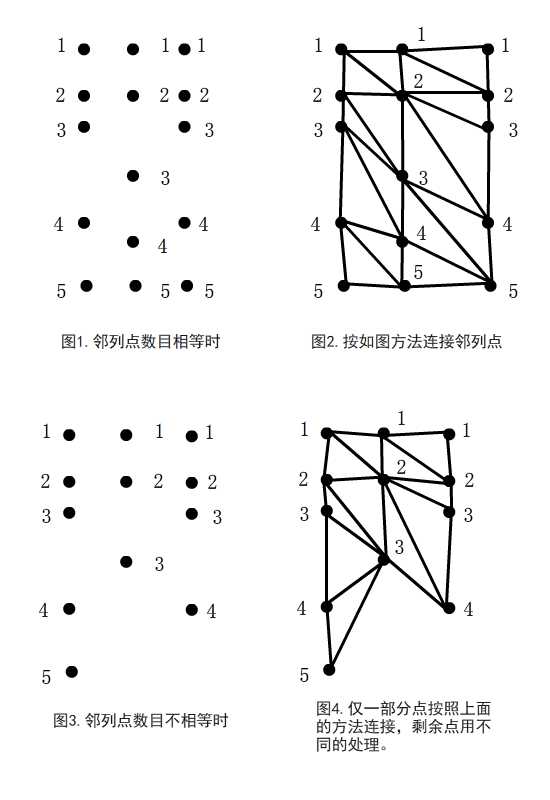
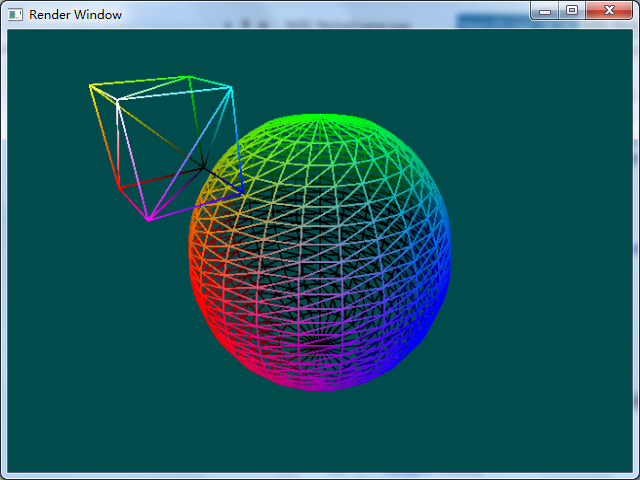
****

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**

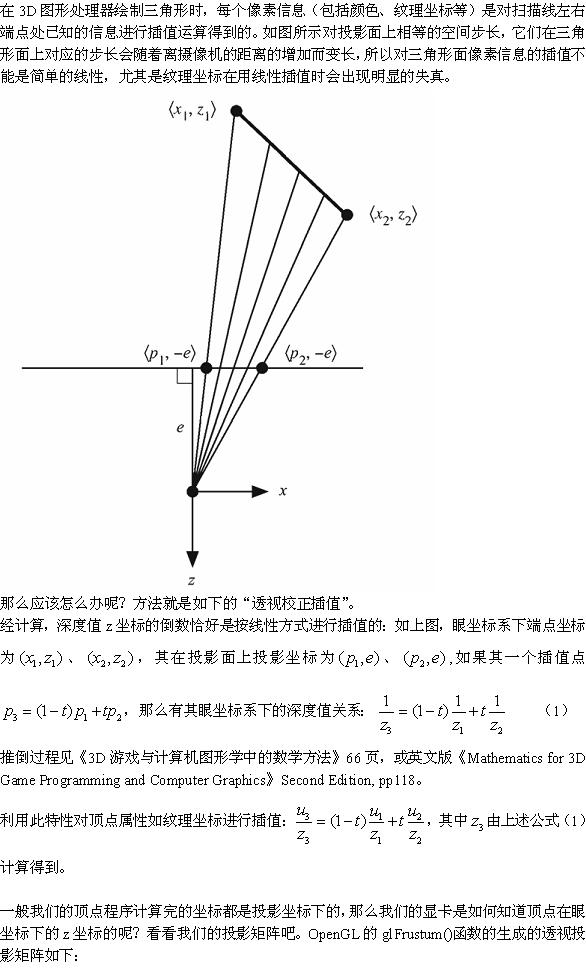
(2015.7.12)暂时是多个光源的三种颜色的叠加：Ambient环境光，Diffuse漫反射，Specular镜面光（高光）。Ambient是程序设定好的作为一个基色。Diffuse是用cos law（Lambertian body），并且有一个mDiffuseIntensity来Scale，这个是view Point independent的颜色，与观察者位置无关，只跟法线和入射光方向有关。Specular是view Point dependent的，与 反射光方向和观察者direction的夹角 有关，并且也有一个mSpecularIntensity来Scale。这个是最基本的模型。三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

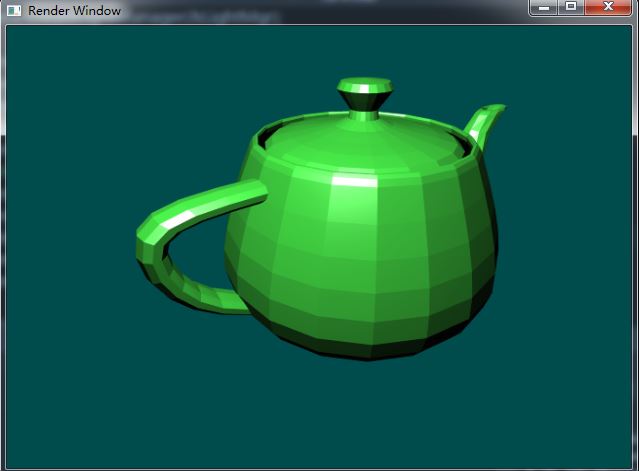
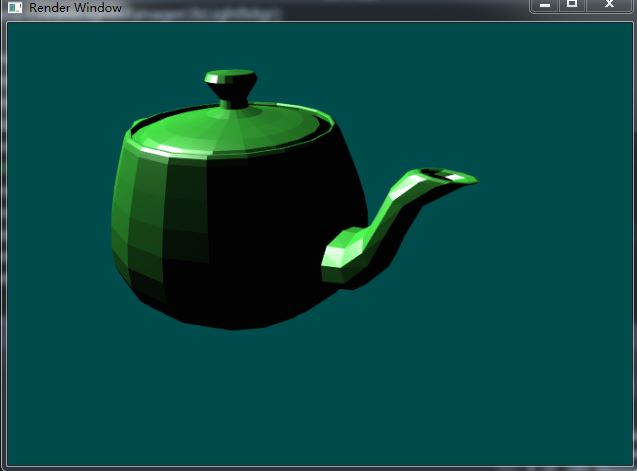
3个4字节浮点数(2顶点的坐标)

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。

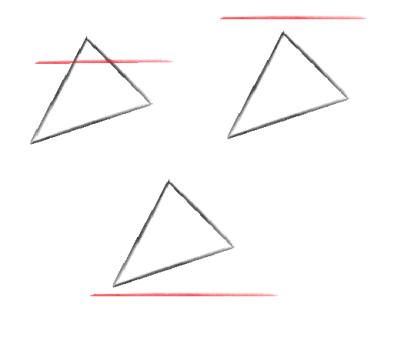
**最后，上图**



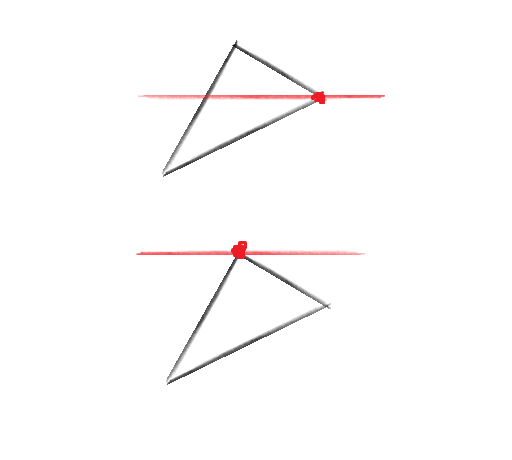
**\*\*\*\*\*\*\*\*\*————2015.7.30————\*\*\*\*\*\*\*\*\*\*\***

**NoiseUtSlicer**

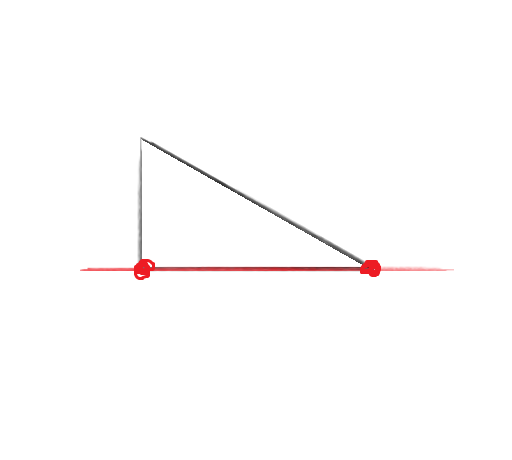
**3D打印机切层器**

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. 

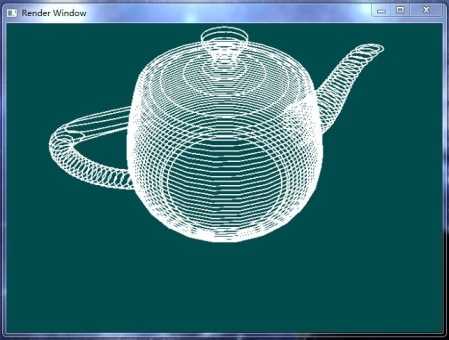
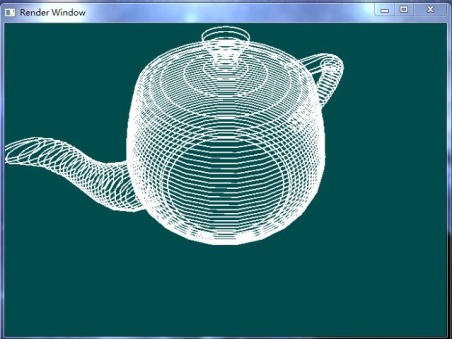
0个顶点在切层上，只能尝试用边去切割

1. 

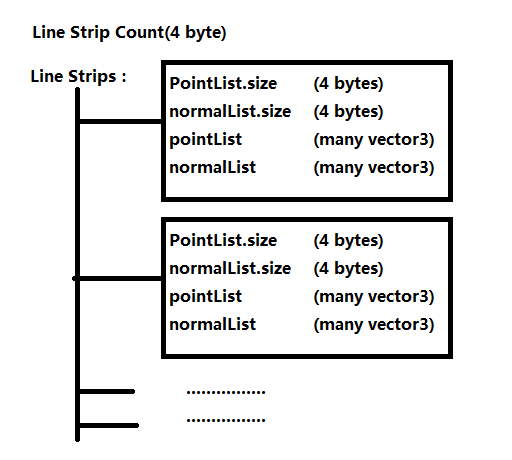
1个顶点在切层上，看看可不可能切出第二个顶点

1. 

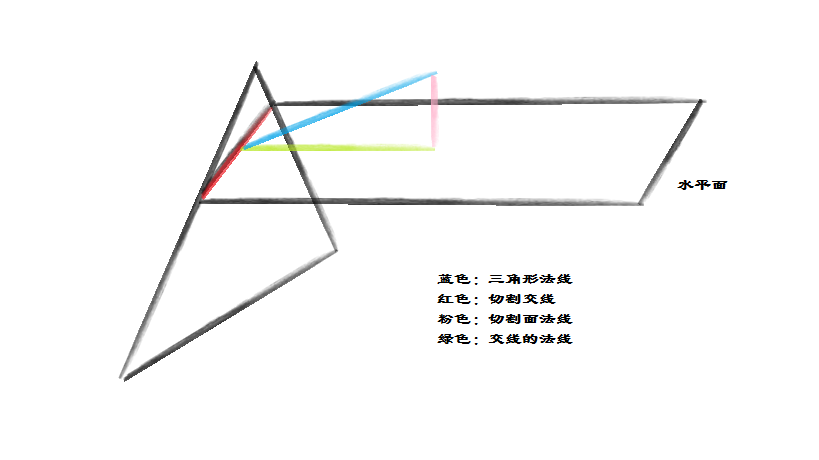
两个顶点在切割面上，直接就是想要的线段。

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

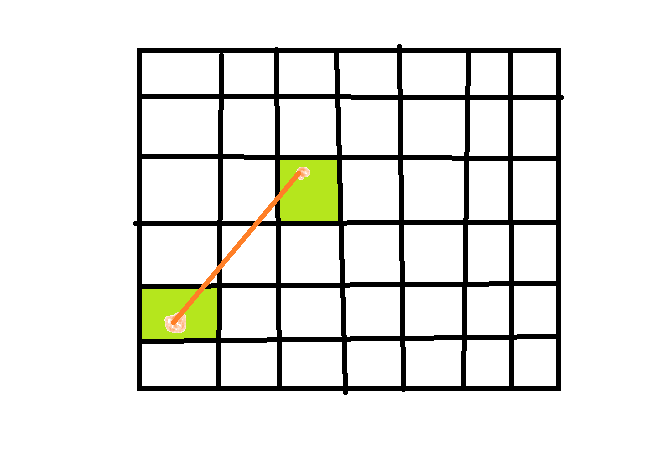
∵

∴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

1. (2015.9.30)连接线段算法：对于每一条正在处理的line strip，查找所有dirty标记为false的线段，看看v1或者v2能不能焊接上line strip的尾部，如果可以就在linestrip的pointlist和normalist上增加东西，然后更新当前的tail point。不过这样非常慢，因为对于每一个line strip的tail point要查找能焊接的顶点基本都要遍历所有line segment,时间复杂度直逼O(n^2)，而且n一般非常大，这是不能接受的。所以现在做一定的预处理改进：新增Layer数据结构，layer被分成很多个tile（或者说是块），每个tile都有std::vector来装line segment的ID（相当于先预先计算好每条线段所属layer的所属tile）。只要有至少一个顶点在当前tile，这条线段就属于这个tile。这么分块是因为，有可能用于焊接到line strip末尾的顶点，必定在附近，所以遍历line segment只需要在tail point所在layer tile搜寻即可。记得焊接好顶点后，把新顶点所属line segment的信息从layer tile的列表中删去(或者点亮line segment的dirty标记），从此这条line segment不再属于这个tile。



如图：红色线段同时属于绿色的layer tile，如果line segment已经加入了line strip，那么它的信息要从两个绿色块中抹除。

