# 模型网格简化 - 总结

## 第一部分：算法分类汇总

由于计算机在处理大规模的三维网格过程中会出现严重的延迟，影响场景显示的视觉效果。当三维模型网格面片数目过大时，会使得场景的渲染速度过慢，性能上受限。因此，为了尽可能的改善场景在渲染等方面的性能，需要对三维模型网格进行简化，而简化过程中也应该尽可能的保持原模型的特征，以尽可能保证逼真度不会明显下降，这样在计算机渲染速度和场景效果显示这二者中以达到一种最佳的平衡。

在学术界对模型简化这一工作已经有一系列的成果。在这一领域专业术语叫做曲面简化，在保证必要精度的前提下，去除冗余信息，以利于图形显示的实时性、数据存储的经济性和数据传输的快速性。以下是我所看过的各式各类算法的归总：

### 1、近平面合并算法

顾名思义，就是说寻找那些共面或者近似共面的三角形进行合并，然后对合并后的多边形进行重新三角化。重新三角化后的三角形数目会相对原数目减少2-3个，从而达到简化的目的。

### 2、顶点删除算法

顶点删除，很显然就是通过移除某些顶点，然后在移除这个顶点之后所形成的空洞中重新三角化，从而减少面数的目的。那么什么样的顶点应该被移除呢？这个评判标准可能五花八门。比较典型的是，对该顶点所关联的所有三角形计算出一个平均平面（计算方式如下），通过计算该顶点距离该平均平面的距离，用该距离作为该顶点对整个模型的重要度。那么很明显，这个重要度值越小，代表该顶点越不重要，移除它对整个模型的轮廓外形影响越低。

计算平均平面：假设当前考虑顶点为X，那么

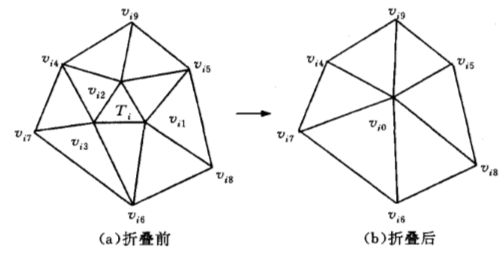
* Ti 为以X为顶点的所有三角形集合
* Ci，Ai，和Ni为Ti中第i个三角形的中心，面积以及法向量
* 那么计算顶点X在平均平面的一个投影点B为：
* 计算出平均平面的法向为：
* 最后有了平面标准表达式ax + by + cz + d = 0，可以很容易计算出点到平面的距离

### 3、边折叠算法

边折叠算法思想主要是以边为简化的单位，按照一定的准则折叠网格模型中的一条边。具体做法就是将选定的2个端点合并为一个点。同时保持缘来网各种顶点之间的连接关系不变。关于边折叠算法后面将详细阐述。

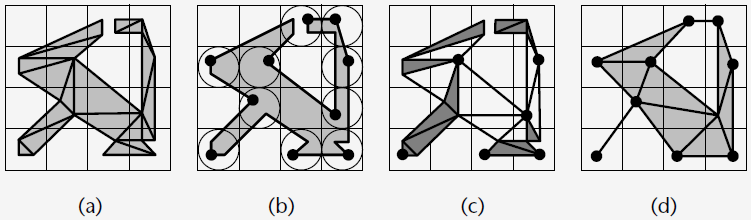
### 4、三角形折叠算法

同以上各类算法，三角形折叠算法以三角形为基本单位，折叠一个三角形到一个新的点，该新的点重新与原三角形三个点关联的顶点相连重新构成新的三角形，这样在三角数量上就减少了。那么根据新点的确定，可以是被折叠三角形的其中一个顶点，也可以是通过某种计算，新生成一个点。三角形折叠大致示意如下图所示：



### 5、顶点聚类算法（细胞折叠）

基本思想是：用一个包围盒把原网格模型包围起来，通过等分包围盒的各棱边将包围盒等分成若干个小的长方体，这样原来模型的所有顶点就分别落在了这些长方体内，然后只要扫描这些长方体，如果某个长方体内有顶点那么就把这个长方体内的所有顶点删除并生成一个新的顶点，这个新的顶点是被删除的原来该长方体内所有顶点的一个加权平均。事实证明，其实这个算法具有很大局限性，即给定任何一个网格模型，我们很难断定该网格模型的顶点分布是均匀的，绝大多数网格的顶点分布都是不确定的非均匀的，这也就意味着当进行等分包围盒时非常容易出现某个包围盒中顶点很多，而某些包围盒顶点却很少，这样顶点很多的包围盒中也同样被聚成单个顶点，最后模型的几何形状就会受很大影响。细胞折叠算法示意图大致为：（来自《Level of Detail For 3d Graphics》）



### 6、重采样算法

这个算法首先在初始网格上任意的插入新的顶点，然后在网格曲面上移动并取代那些曲率最大的位置（关于曲率的衡量标准，后面会谈到）。这样反复的删除初始顶点，最终生成一个由新顶点组成的新的网格。

此外可能还有很多种简化算法，但是这全部算法大致出发点基本都一致：用某种评判准则对几何元素进行删除或者合并等操作，这一操作能够减少面片的数量，以达到简化的目的。

选择对以上算法中比较典型的第（2）和第（3）中进行实现，即顶点删除和边折叠

## 第二部分：几何简化算法实现

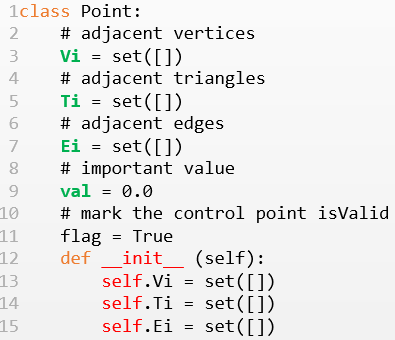
### 1、顶点删除算法

顶点删除算法的算法思想在（2）中已经叙述，算法所涉及的主要数据结构如下：

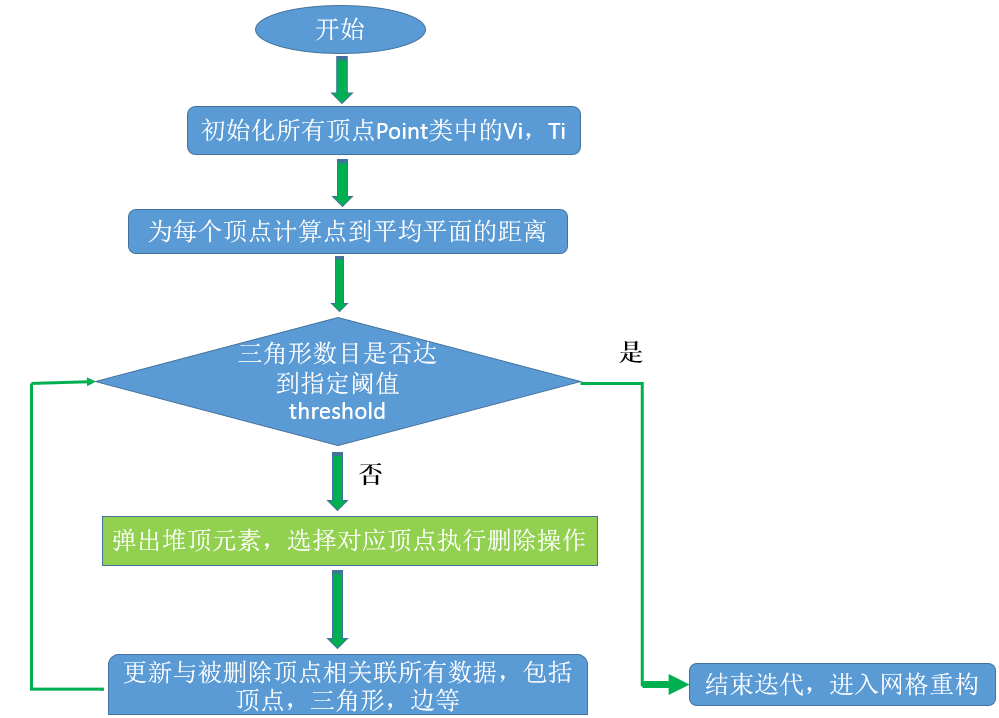
Vi：该顶点所连接的所有相邻顶点的集合

Ti：该顶点所关联的所有三角形的集合

Ei：该顶点所关联的所有边的集合

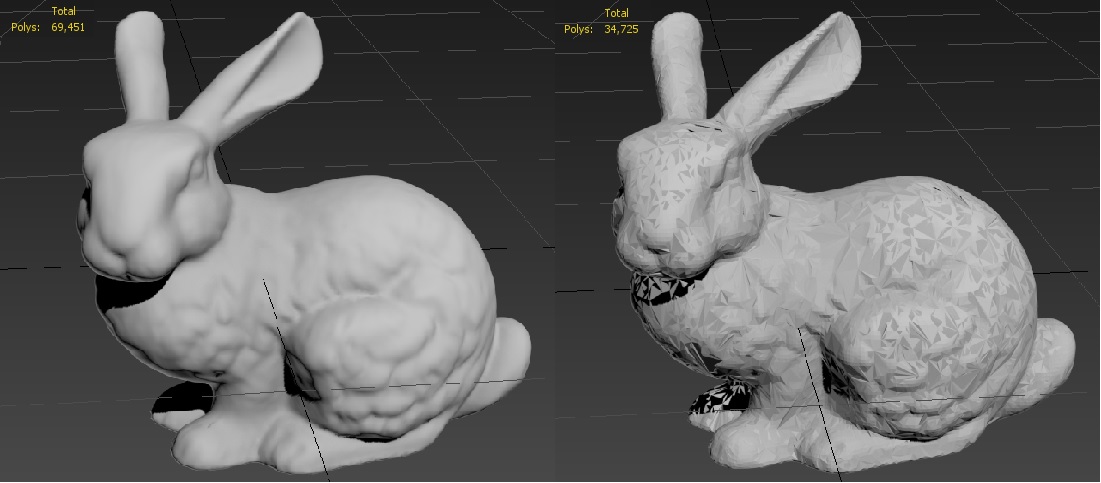
Val：依据该顶点所计算出来的重要度（即顶点到平均平面的距离）

那么当为所有顶点都计算出其重要度之后，将全部顶点的重要度值Val入堆，同时维护好该Val值与顶点的映射关系以便能够找到当前是哪一个顶点。大致流程如下所示：



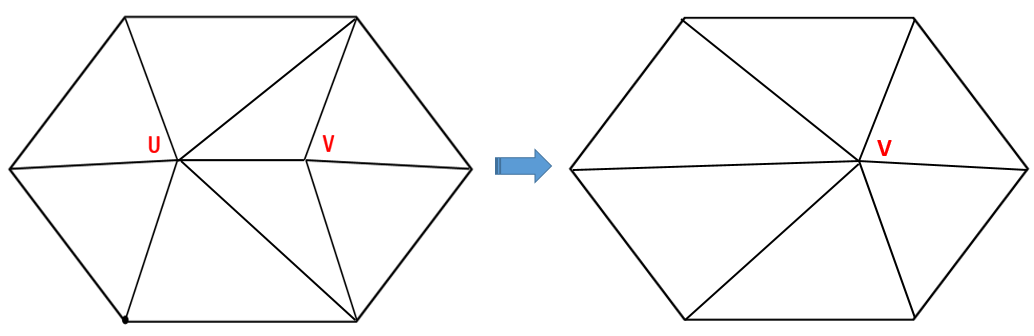
**实验结果：**

用顶点删除算法对斯坦福兔子模型进行试验，下图为简化50%前后对比图：



### 2、边折叠算法

边折叠算法大致思想为：为模型网格中每条边计算一个折叠代价值Cost，依据该Cost值选择最小的边进行折叠，大致示意图如下所示：



显然，当折叠边<u, v>之后，当前区块三角形数由6个减少到4个，因此边折叠算法可以比较稳定的每个轮次降2各个面。

那么边折叠代价值Cost的计算，也同样有着不同的评断标准，这里采用一篇文章中提出的，结合边长度和三角形法向二者相结合的方式来计算边折叠的代价，该文章为《A Simple, Fast, and Effective Polygon Reduction Algorithm》:

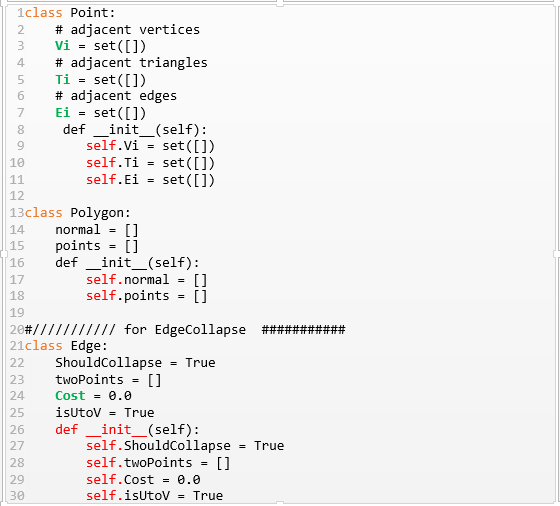


**说明一下：**

* 是边<u, v>的几何长度
* 定义为该条边的曲率元素，几何意义是取u的邻边三角形中，与其余uv的邻边三角形偏离最远的那个三角形，二者法向的夹角（实际上并不是夹角大小，而是衡量夹角大小关系的定量表示）

折叠时，每次选择折叠代价最小的边进行图示中折叠操作。将一条边折叠为一个顶点，整个网格会因此减少与被折叠顶点相关的所有边以及所有三角形，并生成新的若干个三角形，最终三角形数因此而减少2个（大多数情况每折叠一次是两个，除了边缘部分特殊）

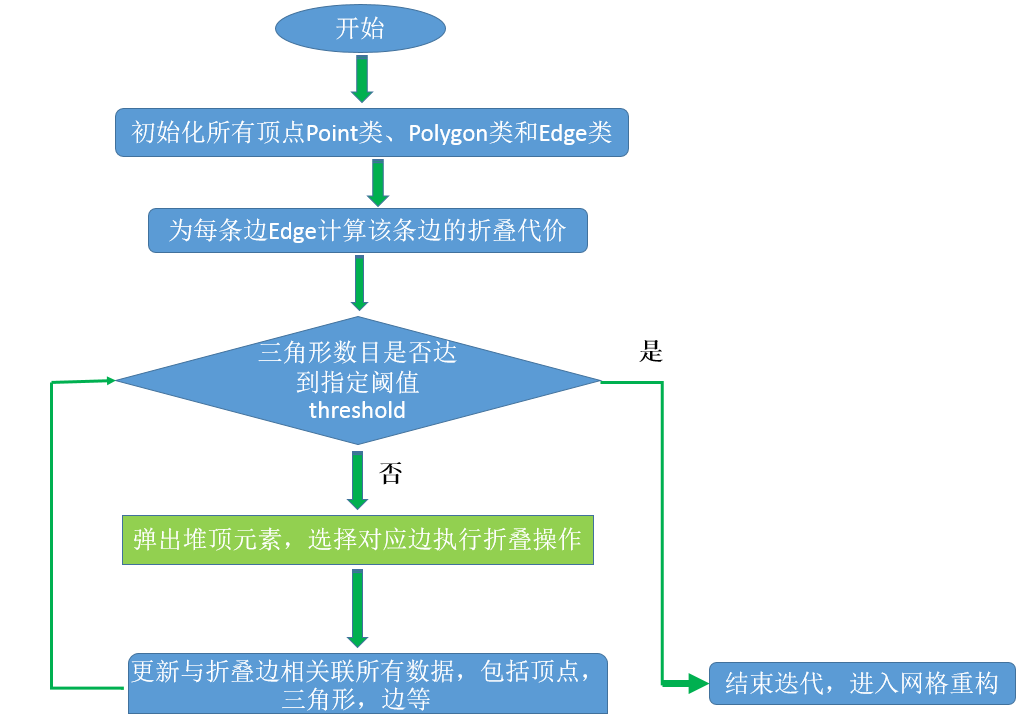
下面是边折叠所涉及的主要数据结构：



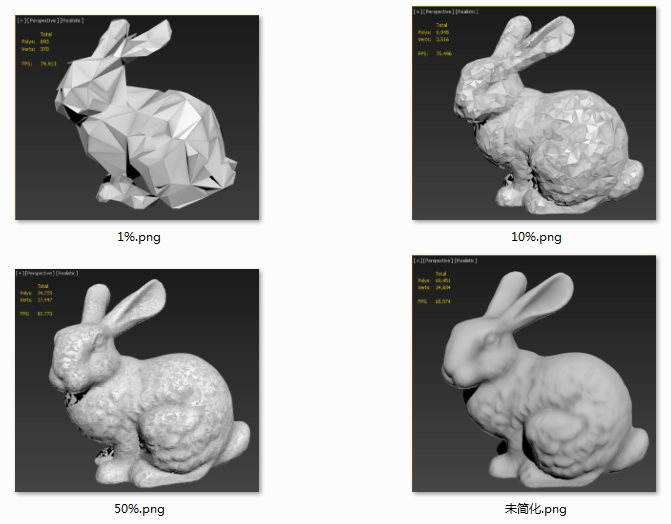
其中：

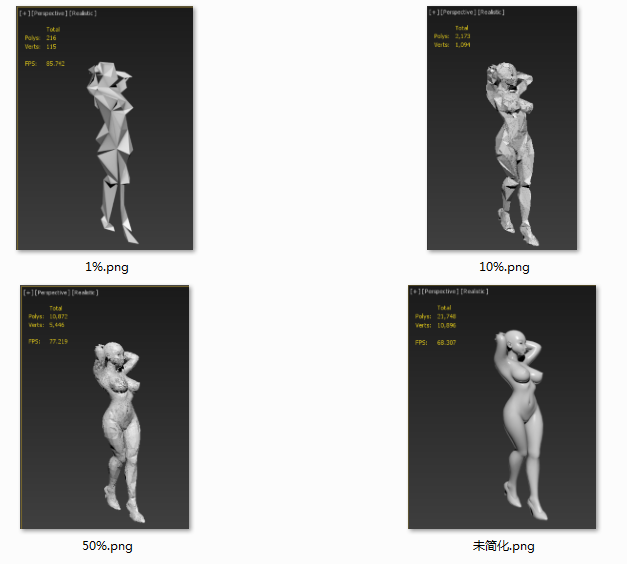
* Point类和上一个功能类似，在此基础上增加一个Ei，表示该顶点所连接的所有边集合，为了折叠后更新数据使用
* Polygon类的设计主要是为了保存所有多边形（三角形）的法向以及三个顶点，以便于快速取出拿来计算每条边的折叠代价，从而避免大量重复计算
* Edge类包含该条边所计算出来的折叠代价Cost，以及边的两个顶点，isUtoV记录该条边的较小的折叠代价时从顶点U -> V还是从V -> U

边折叠大致流程如下所示：



对兔子模型执行边折叠算法效果不错，这里给出边折叠简化的多种层次,如下图所示：





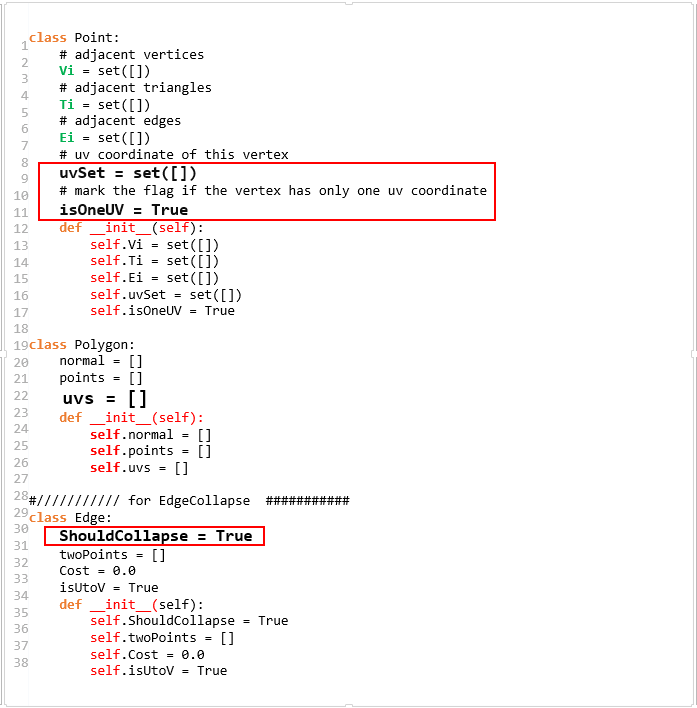
## 第三部分：带纹理属性简化算法实现

前面针对模型的几何特征对模型网格进行简化实际上目标单一，仅仅是为了保持模型的几何形状而进行的简化。但是模型的几何形状并不是唯一重要的，通常还有一些额外的属性，描述模型表面细节特征的比如颜色、纹理等信息。当我们看到一个人，从远处看，可能我们人眼的视觉只能获取到一个人的外形轮廓，这时候可能模型的表面细节特征并不显得那么重要，但是当距离在一定近距时，模型表面的细节能够被人眼所识别，这时候单一以几何形状简化为目标已不能够很好的描述模型，因为丢失了很多模型表面细节，使得逼真度降低。

因此在简化时将模型表面细节属性等考虑进来，有助于得到一个更有利于视觉效果的模型简化层次（所谓的LOD）。所以就模型表面细节特征中的纹理信息作为依据，结合几何简化进行实验。

带纹理模型简化主要存在的问题是单个模型可能具有多张纹理，在不同纹理区域的边界可能会出现比较明显的纹理扭曲或者拉伸，导致细节丢失。那么算法在考虑带纹理的简化时，原则主要是为每条待折叠的边（或者顶点）设置一个标志，若该条边或顶点具有多种不同的纹理映射（u, v），那么保留这样的顶点或者边，只对非纹理边界的边进行折叠或者删除，这样折叠后新生成的面片组合起来所得到的完整区域保证和之前折叠边所关联的顶点组成的区域映射到的纹理区域基本为同一块，因此经过这样的简化之后能够较好的保持纹理信息。

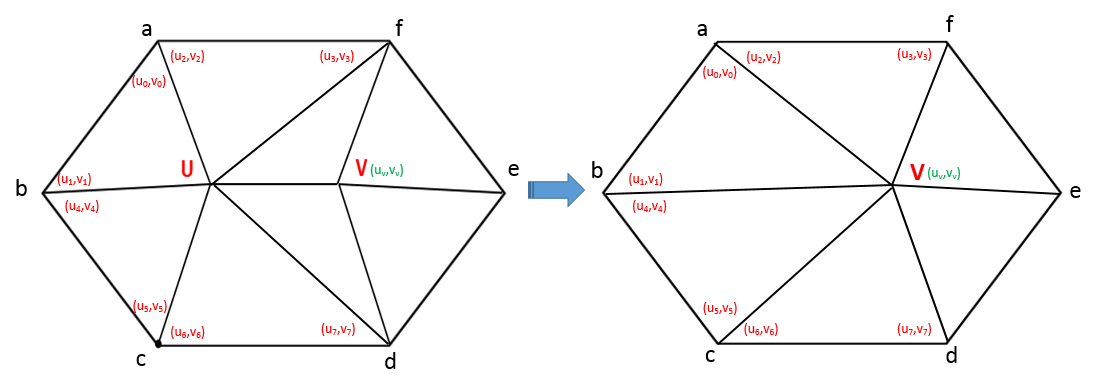
带纹理简化算法的实现采用前面效果较好的边折叠算法，相对于几何简化过程而言，带纹理的模型简化主要增加是在以下几项：



* 在Point类中增加该几何顶点所对应的全部（u，v）纹理坐标，可能有1个，可能有多个。再增加isOneUV标志，表示该几何顶点坐标所对应的UV纹理坐标是否单一，若是，则该顶点可被考虑删除，否则不允许删除（在Edge类中得以体现）
* 在Polygon类中增加与顶点坐标对应的（u，v）纹理坐标，初始化时一并读取纹理坐标信息
* 在Edge类中增加ShouldCollapse标志，表示如果该条边的两个顶点若有其中一个顶点包含多重uv纹理，则不折叠该条边

带纹理时，执行边折叠操作局部操作如下所述：

其中点U、V两个顶点均具有单一纹理坐标，当折叠U至V时，纹理坐标的变换为图中所示：



假设顶点U的邻接顶点集合为Vu，顶点V的邻接顶点集合为Vv，那么在折叠<u, v>这条边之后，三角化所形成的空洞时，即让Vu - Vv的顶点连接至V，其中<a, b>,<b, c>, <c, d>,<a, f>这几条边中两个点的纹理坐标保持为其原先所在三角形中的纹理坐标，那么经过三角化所生成的新的三角形，构成的新的三个点的纹理映射坐标：

三角形（a，b，v）：(u0,v0), (u1,v1),(uv, vv)

三角形（a，f，v）：(u2,v2), (u3,v3),(uv, vv)

三角形（b，c，v）：(u4,v4), (u5,v5),(uv, vv)

三角形（c，d，v）：(u6,v6), (u7,v7),(uv, vv)

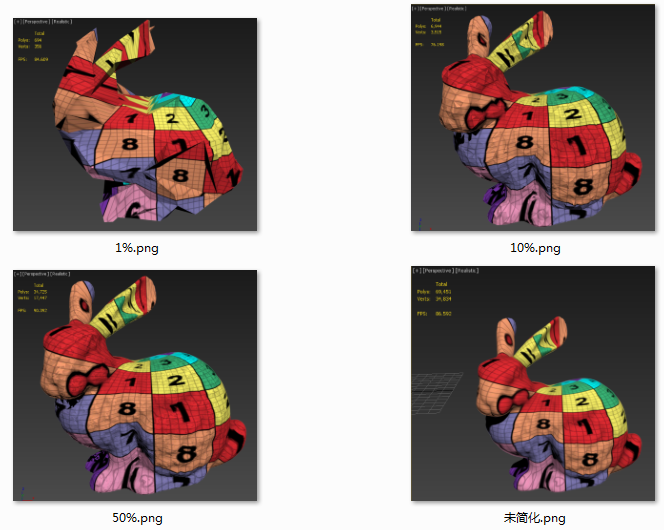
最终由新生成的四个三角面所连接的纹理将整体显示出原6个面的纹理，因此可以较好地保持纹理信息

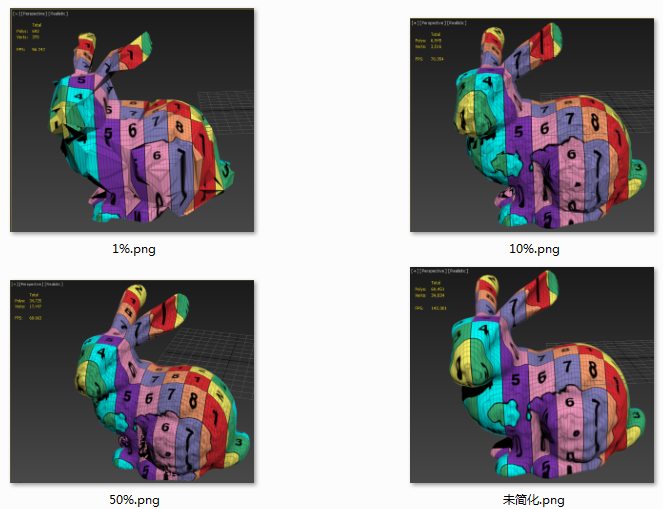
带纹理的简化与几何简化在最后的网格重构时，有些区别：

* 为节点创建新的mesh属性，这点和几何简化步骤相同
* 为新创建的mesh属性创建新的UV元素，并设置映射模式为：
  + 映射模式 FbxLayerElement.eByPolygonVertex
  + 访问模式 FbxLayerElement.eIndexToDirect
* 复制原来mesh属性纹理坐标数组
* 最终在重建三角网格时同时设置每个三角形每个顶点所对应的访问下标，这样就维持了多边形顶点和纹理坐标的映射关系

最后有一点就是我感觉大多数软件所生成的fbx文件对控点的存储都具有很大冗余，对于控点而言，本身在三维空间中的坐标是唯一的，如果每个控点都重复多个可能会造成文件大小膨胀。不过这也可能是因为软件为了保证与其他软件的兼容性而这么做的。在最终的重构时，我去除了控点数组中全部冗余的控点，每一个三维坐标在控点数组中有且仅有一份拷贝。

对带纹理的兔子模型执行简化，对两组不同纹理贴图的兔子模型实验结果截图如下所示：





## 第四部分：在MFC下用DirectX将算法结果以界面形式表现出来，形成简单的小工具

本次任务主要目标是熟悉D3D的渲染管线的流程，那么其中最主要的也是顶点处理阶段，即模型的各种空间的变化。借助这些变化完成界面上鼠标的拖拽平移和旋转操作。最后分屏显现出来，将简化前和简化后的模型形成一个鲜明的对比效果。

### 1、3D中有如下三种空间

#### 1）模型空间

　 模型空间也叫本地空间或建模空间，这是我们定义物体三角形的坐标系，当一个美工人员创建一个物体的三维模型的时候，他选择了一个方便的方向，比例和位置来保存模型的组成顶点，物体的模型空间和其他模型没用任何关系。

#### 2）世界空间

　 上面我们提到过，物体的模型空间和其他物体的模型空间是没用任何关系的，而世界空间的主要目的就是给你的场景中的物体提供一个绝对的参考。

#### 3）视角空间

　 这个坐标是以摄像机为基准的，以摄像机的位置为原点，摄像机朝向Z轴正方向，右边为X轴正方向，上边为Y轴正方向，之所以设置这个坐标系，主要是为了方便投影及裁剪操作。

#### 4）投影，裁剪空间

　 这个空间即世界空间的物体被投影到相应的投影面上之后，继而进行的裁剪操作所在的空间。

### 2、主要有以下几种空间之间的变换

#### 1）模型空间------>世界空间

　 建模变换： 在物体空间中指定的物体被放置到世界空间的方法要依靠建模变换。如：你将一张桌子放置的一个房间里面，你需要旋转，平移，缩放等操作，才能使桌子按照你的要求放置的房间的准确位置，在同一间房间里面我们可以放置同样的桌子模型，使用不同的建模变换，可以使桌子在房间的不同位置，不同的方向放置。

#### 2）世界空间------>视角空间

视角变换：从世界空间到视角空间通过相应的视角矩阵来实现，所谓视角空间，我们可以这里理解，就是以摄像机为原点，由视线方向，视角和远近平面共同组成的一个梯形体。也被称之为视锥体，其由近平面和远平面组成。近平面，是梯形体较小的矩形面，同时也是靠近摄像机的平面。远平面，就是梯形体较大的矩形，作为投影平面，在这个梯形体的内部的数据是可见的，超出部分会被剔除，也叫视锥裁剪。在游戏中，屏幕的内容随摄像机的移动而变化，是因为GPU将物体的顶点坐标从世界空间转换为视角空间。

#### 3）视角空间------>投影、裁剪空间

投影变换：从视角空间到投影，裁剪空间依靠投影矩阵来实现，投影有两种：正交投影和透视投影，在大多数游戏中用到的都是透视投影，因为这种投影方式与人观察物体的方式是一致的这个过程其实就是一个投影，裁剪，映射 的过程，因为在一个不规则的视锥体内裁剪是一件非常困难的事情，所以将裁剪安排在一个单位立方体中进行，这个立方体被称为规范立方体。关于投影变换，要注意的一点是，很多人误以为投影即是把三维顶点投影到二维平面上，投影变换后顶点的Z坐标被抛弃，只剩下X，Y坐标用于后面屏幕变换，实际上，投影变换后Z坐标并没有消失，位于[0,1]之间，屏幕坐标的变换不再使用Z坐标，但Z坐标在后面的输出管理阶段用于深度比较时会发挥重要作用。

#### 4）投影、裁剪空间------>屏幕空间

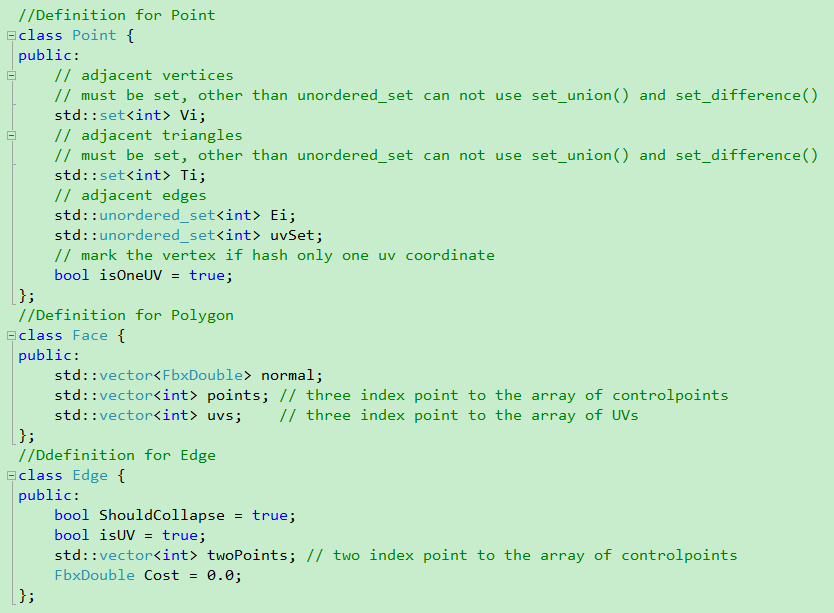
　 视图变换：最后一步是取每个顶点的标准化的设备坐标，然后把它们转换为使用像素度量X和Y的最后的坐标系统，它为图形处理器的光栅器提供数据。然后光栅器从顶点组成点，线段或多边形，并生成最后图像的片段。另一个被称为深度范围变换的变换，缩放顶点的Z值到在深度缓冲中使用的深度缓存的范围内。

（部分摘录自一篇技术博客：<http://www.cnblogs.com/He-Jing/p/3760106.html>）

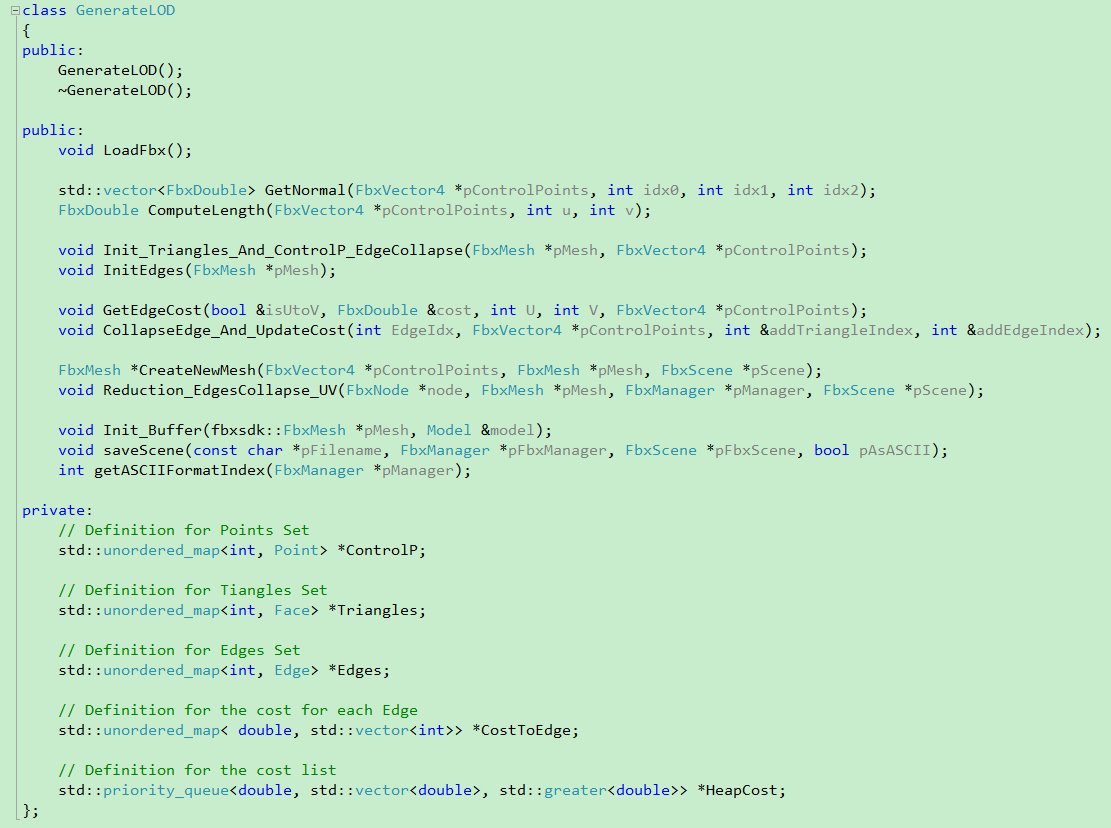
### 3、在C++下主要数据结构的设计

类似在Python下，在C++下数据结构进行类似的设计

#### 1）点、面、边的数据类型

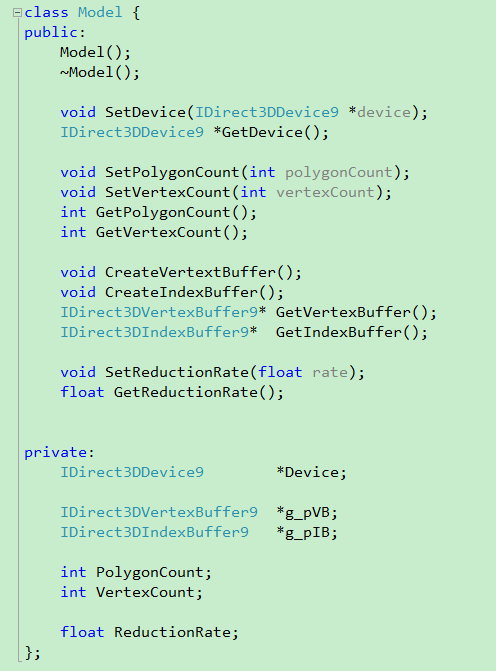


#### 2）算法简化类的定义

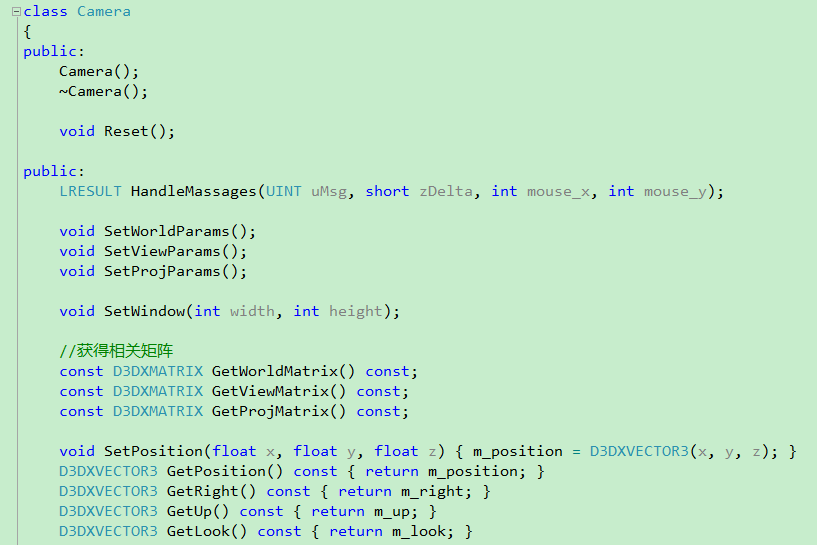


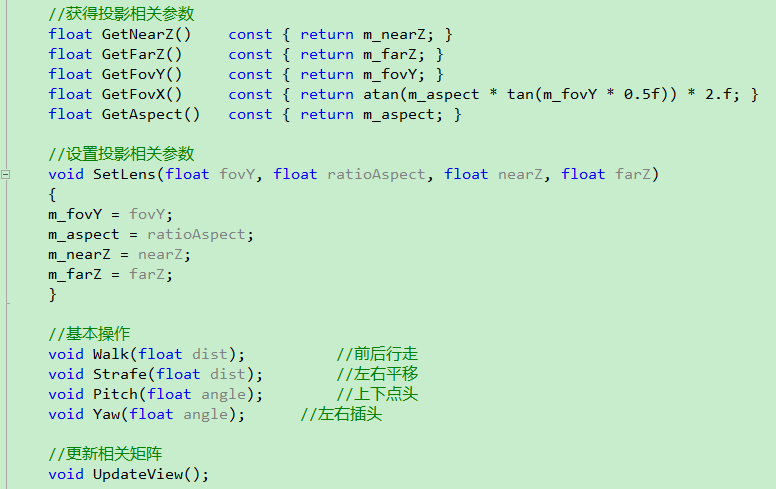
#### 3）D3D中，绘制一个图形所需要的数据

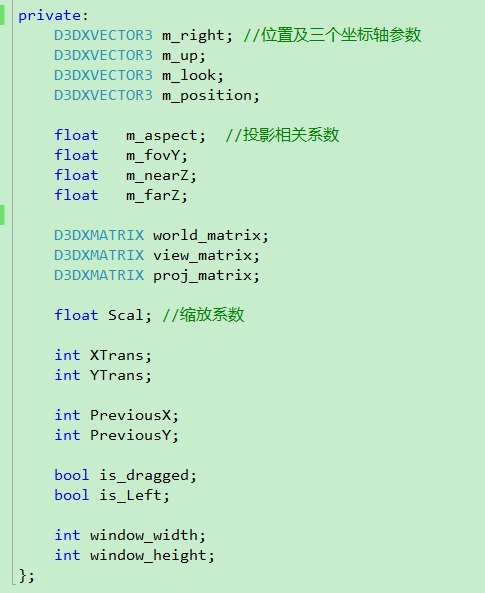
将所需绘制的每一个图形抽象成一个模型，其数据主要由顶点缓存和顶点索引缓存组成，主要数据结构如下：



#### 4）Camera类的设计







说明：Camera类的设计，主要是存储Camera的位置m\_position，以及向右的向量m\_right，向上的向量m\_up，摄像机朝向向量m\_look，以及世界变换矩阵、视角变换矩阵、投影变换矩阵。这三大矩阵在绘制时需要设置矩阵变化时可以直接从Camera类中取得。

保存m\_position、m\_right、m\_up以及m\_look几个位置或向量，这里并非可以直接为辅助函数D3DXMatrixLookAtLH所用，因为辅助函数D3DXMatrixLookAtLH所需要的第三个参数并不是一个向量，而是一个位置坐标，即target目标位置。因此在生成视角变换矩阵时，如果需要借助辅助函数D3DXMatrixLookAtLH生成的话，必须在此之前计算出target位置坐标，计算方式很简单，即用相机位置m\_position在m\_look向量方向上前进若干距离即可。当然，也可以直接通过UpdateView函数直接设置所需要生成的视角变化矩阵响应的行与列。

#### 5）最后就是Direct 3D初始化类的定义



### 4、遇到的问题总结

#### （1）算法用C++运行速度十分缓慢，几乎无法运行

原因：VS下的Debug编译式和Release编译方式的区别

其中Debug编译方式为调试版本，包含调试信息，并且基本不做任何优化。 而Release编译方式为发布版本，进行了最大程度的优化，使得程序在代码大小和运行速度上都是最优的。当时没想到这个原因，感谢领新老师亲自为我调试找出了这个原因。

#### （2）读取出网格数据后绘制出来的图形显示效果十分畸形，箭头式的在边缘扩张

原因：当时因为在绘制左半屏原始模型时顶点缓存只是粗糙的仅使用顶点缓存先测试一下，后面未改过来（并未结合顶点索引），而绘制右半屏简化后的模型时改为了顶点缓存+顶点索引的方式，结果两种绘制方式混合就导致这个结果。后面统一使用顶点缓存+顶点索引的方式绘制，显示正常。

#### （3）两个关于纹理的问题

##### 1）在启用纹理之后，显示的纹理表面细节支离破碎

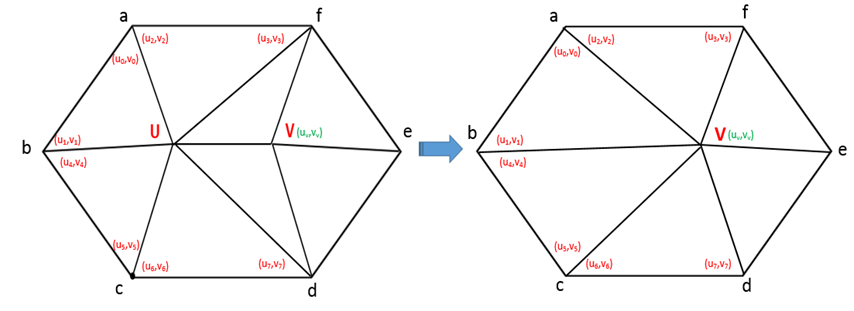
原因：是因为一直被设置的背面消隐模式 D3DCULL\_NONE ，而 D3DCULL\_NONE 表示完全禁用背面消隐，此时显示出的纹理则与背面纹理交叉，导致混乱。

经过测试，FBX文件中的顶点顺序存储方式是为顺时针的（这点同DX一样），因此设置背面消隐模式为D3DCULL\_CCW，表示只对逆时针绕序的三角形进行消隐。此时原始模型纹理显示正常。

##### 2）在1）中的问题解决过后，右半屏的简化之后的模型表面纹理显示依然错综混乱

此时原因已经很明显了：之前在未考虑三角形顶点存储顺序纹理时执行简化算法，如果只显示网格模式的确没问题，但是当启用纹理时就会出错，因为在算法简化过程中，对边折叠或顶点删除操作导致的空洞进行重新三角化时，新增加的三角形顶点顺序没有被考虑，只是单一的三个点构成一个三角形进行存储。

因此在对边折叠或顶点删除操作导致的空洞进行重新三角化时，保持与相关联的三角形顶点顺时针顺序不变，进行重新构造新的三角形的三个顶点，此后简化后的模型纹理上也显示正常。具体如下：



**例如：因为原模型中读取出的三角形三个顶点的存储顺序有以下三种情况，即顶点 U 分别位于 0、1、2位置上：**

* 若原左边局部图形中，三角形<U， b， a>为顺时针方向，则右边局部图形中，新增加的三角形顶点顺序应为<V，b，a>时，才能保证顺时针方向，否则错误
* 若原左边局部图形中，三角形<a， U， b>为顺时针方向，则右边局部图形中，新增加的三角形顶点顺序应为<a，V，b>时，才能保证顺时针方向，否则错误
* 同理，若原左边局部图形中，三角形<b， a， U>为顺时针方向，则右边局部图形中，新增加的三角形顶点顺序应为<b，a，V>时，才能保证顺时针方向，否则错误

#### （4）平移

当左键鼠标在屏幕上左右移动时，实际上是沿着以相机为原点的视角空间向右方向的“X”轴作平移，但这个X轴并不是世界空间的X轴，而是以相机位置作为原点的视角空间的m\_right方向。因为本质上移动的就是相机m\_position的坐标位置，移动方向是m\_right，所以左右移动动作应该是：m\_position += dist \* m\_right;

当左键鼠标在屏幕上上下移动时，实际上是沿着以相机为原点的视角空间向上方向的“Y”轴作平移，但这个Y轴并不是世界空间的Y轴，而是以相机位置作为原点的视角空间的m\_up方向。因为本质上移动的就是相机m\_position的坐标位置，移动方向是m\_up，所以上下移动动作应该是：m\_position += dist \* m\_up;

#### （5）旋转

当右键鼠标在屏幕上左右移动时，实际上是在绕着以相机为原点的视角空间的“Y”轴旋转，但这个Y轴并不是世界空间的Y轴，而是以相机位置作为原点的视角空间的m\_up方向。所以这里的第一个误区就是不能使用D3DXMatrixRotationY函数生成旋转矩阵，而要使用D3DXMatrixRotationAxis函数绕m\_up向量生成旋转矩阵，然后用这个旋转矩阵去更新m\_right和m\_look向量。更新完这两个向量之后，还有一个非常重要的地方，也就是第二个误区，同时也要更新相机位置m\_position的坐标。因为绕m\_up旋转唯一不变的也只是这个m\_up向量，其他的都在绕着它旋转.所以必须要：

* D3DXVec3TransformCoord(&m\_position, &m\_position, &rotation);

当右键鼠标在屏幕上上下移动时，实际上是在绕着以相机为原点的视角空间的“X”轴旋转，但这个X轴并不是世界空间的X轴，而是以相机位置作为原点的视角空间的m\_right方向。同上，所以这里的第一个误区就是不能使用D3DXMatrixRotationX函数生成旋转矩阵，而要使用D3DXMatrixRotationAxis函数绕m\_right向量生成旋转矩阵，然后用这个旋转矩阵去更新m\_up和m\_look向量。同上，更新完这两个向量之后，还有一个非常重要的地方，也就是第二个误区，同时也要更新相机位置m\_position的坐标。因为绕m\_right旋转唯一不变的也只是这个m\_right向量，其他的都在绕着它旋转。

最后，也就是最重要的一点，也就是第三个误区，我们存储的一个向量是朝向m\_look，这个是朝向，而不是目标target，而辅助函数D3DXMatrixLookAtLH所需要的第三个参数是目标target位置，这个target位置是第二个参数相机位置（即m\_position）沿着这个m\_look向量向前的某个位置：

* D3DXVECTOR3 m\_LookAt = m\_position + m\_look;
* D3DXMatrixLookAtLH(&view\_matrix, &m\_position, &m\_LookAt, &m\_up);

当然，也可以在相机Camera类中直接存储 视点（Eye）+ 目标（target） + 向上（Up），然后每次更新时直接更新这个三个点或者向量，这样就能直接为生成视角变换矩阵的函数D3DXMatrixLookAtLH所用了。但是为了直观的好理解，代码中使用存储m\_look向量。两种方式原理是一样的。

另外，代码中也实现了另一个函数：UpdateView()函数，这个函数等同于D3DXMatrixLookAtLH函数的功能，直接把三个向量：m\_right、m\_up、m\_look存储到视角变换矩阵的相应行与列中。经过试验，与辅助函数D3DXMatrixLookAtLH的效果一致。

## 第五部分：一点感想

在首次接触到3D模型，纹理贴图等这些基本概念时有着太多的疑问，但是随着努力的查阅资料，进行各种搜索学习之后，受益良多。也似乎找到了在学校里研究方向没有的乐趣。后期在学习Direct3D 时也是受益匪浅，熟悉了Direct3D 的整个大致流程。期间碰到问题，并且思考解决问题，这个过程是个宝贵的过程。尤其是在初期学习3D各种空间变换时，在代码实现时遇到的各种问题，只是很多时候把问题想象的太难了，主要还是需要静下来把原理从头到尾理清楚，解决问题才会比较有头绪。另外在领新老师指导下，意识到实现一个功能是一回事，把一种实现整理成易于维护、易于理解、条理清晰、功能模块比较独立的工程性代码很重要，这样的代码自己看着也舒服，看得人也比较容易理解在做什么，分别是怎么做的。这两个月以来意识到自己的问题还真不少，希望自己以后在这些方面能够多加改善！不断进步！