简介

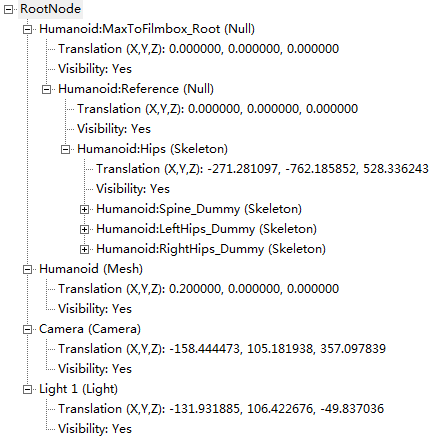
FBX是Autodesk的一个用于跨平台的免费三维数据交换的格式（最早不是由Autodesk开发，但后来被其收购），目前被 众多的标准建模软件所支持，在游戏开发领域也常用来作为各种建模工具的标准导出格式。Autodesk提供了基于C++（还有[Python](http://lib.csdn.net/base/11)）的SDK来实现对FBX格式的各种读写、修改以及转换等操作，之所以如此是因为FBX的格式不是公开的，这也是FBX的诟病之一。与FBX相对的则是格式开源的Collada，它的应用也很广泛。总体来说这两种格式还是各有优劣不相上下，关于两种格式在游戏开发中使用的对比与讨论也比较多，可见GameDev中的帖子：<http://www.gamedev.net/topic/467753-collada-vs-autodesk-fbx> , 这里就不再论述了。大多数情况下我们是需要解析模型在程序中渲染使用，因此这里主要讨论使用FBX SDK来对FBX模型进行解析与加载（主要包括几何网格、材质、Light与Camera、Skeleton动画等），而对于在建模工具中可能涉及到的FBX写出等则没有涉及。

2. FBX SDK的配置

首先，在使用前需要下载安装FBX的SDK，可以从Autodesk的网站上进行获得最新的版本[http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=7478532](http://usa.autodesk.com/adsk/servlet/pc/item?id=10775855%5C&siteID=123112)(需要填些基本信息注册下)。安装之后在VS里边的配置就跟D3D类似。其中的Samples基本上涵盖了FBX相关的应用，可以在使用之前好好研究一下。最新的SDK版本（2012版）与之前的版本会在细节上有所不同（有些较大的改动是实现某些功能 的API接口的修改，具体细节可以用2012的Programmer's guide中找到），而且支持最新的FBX格式，因此最好就直接用最新的版本。

3. FBX模型的组织结构

FBX是以scene graph的结构来存储模型的所有信息（也可以认为是一个多叉树），类似于OSG中的组织方式，这一点可以从SDK所带的Sample里边很清楚的看出来。一个较为典型的模型的组织结构与下图所示：



整个Scene是从一个空属性的根结点开始，其中每个结点均是一个KFbxNode的对象，所有对象之间的关联均是双向的，比如从子结点可以索引到父结点，从父结点也可以索引到子结点；从单个结点可以索引到整个Scene，从Scene也可以索引到该结点。每个结点都会有一个标记属性的Enum值，比如eMesh、eLight、eCamera或eSkeleton等，分别用来标记当前结点是Mesh、Light、Camera或Skeleton。在整个结构的遍历过程中可以通过判断不同的结点属性而进行不同的处理操作。

在进行使用SDK进行FBX的处理操作之前需要先初始化两个必须的FBX对象：KFbxSdkManager和KFbxScene。前者用来对所有的FBX对象进行内在管理，所有使用SDK加载的资源均在此对象的管控之下，最终的资源释放也由其来完成。有了内存管理器之后再在其上创建一个相关的KFbxScene对象之后即可以进行模型的加截与处理了。KFbxScene其实相当于Manager提供的整个场景对象的一个入口。两个对象的初始化与配置代码如下所述：

初始化SDKManager

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7210511#)

1. **bool** FBXImporter::Initialize()
2. {
3. // Create the FBX SDK Manager, destroy the old manager at first
4. **if**(mpFBXSDKManager)
5. {
6. mpFBXSDKManager->Destroy();
7. }
8. mpFBXSDKManager = KFbxSdkManager::Create();
10. **if**(mpFBXSDKManager == NULL)
11. {
12. **return** **false**;
13. }
15. // Create an IOSettings object
16. KFbxIOSettings\* ios = KFbxIOSettings::Create(mpFBXSDKManager , IOSROOT);
17. mpFBXSDKManager->SetIOSettings(ios);
19. // Load plug-ins from the executable directory
20. KString lExtension = "dll";
21. KString lPath = KFbxGetApplicationDirectory();
22. mpFBXSDKManager->LoadPluginsDirectory(lPath.Buffer() , lExtension.Buffer());
24. // Create the entity that hold the whole Scene
25. mpFBXSDKScene = KFbxScene::Create(mpFBXSDKManager , "");
27. **return** **true**;
28. }

FbxScene的初始化

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7210511#)

1. **bool** FBXImporter::LoadScene(**constchar**\* pSeneName)
2. {
3. **if**(mpFBXSDKManager == NULL)
4. {
5. **return** **false**;
6. }
8. // Get the file version number generate by the FBX SDK.
9. KFbxSdkManager::GetFileFormatVersion(mSDKVersion.mMajor , mSDKVersion.mMinor , mSDKVersion.mRevision);
11. // Create an importer.
12. KFbxImporter\* pKFBXImporter = KFbxImporter::Create(mpFBXSDKManager ,"");
14. // Initialize the importer by providing a filename
15. FBXFileVersion fileVersion;
17. **bool** importStatus = pKFBXImporter->Initialize(fileName , -1 , mpFBXSDKManager->GetIOSettings());
18. lImporter->GetFileVersion(fileVersion.mMajor , fileVersion.mMinor , fileVersion.mRevision);
20. **if**(!importStatus)
21. {
22. **return** **false**;
23. }
25. // Import the scene
26. mpFBXScene->Clear();
28. importStatus = pKFBXImporter->Import(m\_mpFBXScene);
30. // Destroy the importer.
31. pKFBXImporter->Destroy();
33. **return** importStatus;
34. }

在完成了对KFbxScene的初始化操作之后即可以从其中得到整个模型的所有信息。由于FBX的是采用了类似于树的结构来进行存储，因而就很容易使用类似于树的递归方法来遍历其中的每个结点，并根据结点的属性选择合适的处理操作，下述代码就完成了从根结点开始的全局遍历：

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7210511#)

1. **void** ProcessNode(KFbxNode\* pNode)
2. {
3. KFbxNodeAttribute::EAttributeType attributeType;
5. **if**(pNode->GetNodeAttribute())
6. {
7. **switch**(pNode->GetNodeAttribute()->GetAttributeType())
8. {
9. **case** KFbxNodeAttribute::eMESH:
10. ProcessMesh(pNode);
11. **break**;
12. **case** KFbxNodeAttribute::eSKELETON:
13. ProcessSkeleton(pNode);
14. **break**;
15. **case** KFbxNodeAttribute::eLIGHT:
16. ProcessLight(pNode);
17. **break**;
18. **case** KFbxNodeAttribute::eCAMERA:
19. ProcessCamera();
20. **break**;
21. }
22. }
24. **for**(**int** i = 0 ; i < pNode->GetChildCount() ; ++i)
25. {
26. ProcessNode(pNode->GetChild(i));
27. }
28. }

上述代码比较简单，直接传入由KFbxScene中获得的根结点之后即可遍历到每一个子结点。在FBX的存储中，每个父结点可以包含多个子结点，但每个子结点只有一个根结点，而且这其中的联系是双向的，这样很方便，比如在处理Skeleton时就常常需要从子结点中得到父结点的matrix等信息，而这种双向关系使得这些操作很容易实现。注意，上述代码中有pNode->GetNodeAttribute()检查操作是必须的，因为并不是所有的结点都有相应的属性（Attribute也是以子结点的方式关联到当前的结点上的，因而可能为空）。

4. 加载几何网格

FBX对几何网格支持得还是很好的，Nurbes、Polygon、Triangle等均可以存储。不过为了简化加载和处理时的操作，最好直接在FBX导出插件中选择一种统一的模式。比如可以在导出生成FBX时选中Triangluation的属性，那么FBX导出插件会自动把所有的Nurbes、Polygon三角化为三角形进行存储。当然，这个过程也可以在模型进行加载时来进行。这样在得到的FBX中就只有三角形这样一种网格模型，方便了加载的操作。模型的几何数据主要包括以下部分：

* Vertex       组成网格的顶点信息，这一部分是必须的。
* Color         每个顶点的颜色，一般不需要。
* Normal     每个顶点所对应的法向，是由FBX导出插件计算生成，可以是逐面片或逐顶点。
* UV             每个顶点所对应的法向，是由FBX导出插件计算生成，可以是逐面片或逐顶点。
* Tangent   每个顶点所对应的贴图UV值，一般来说，每个UV对应一个Layer，一个顶点可以有多个UV通道，这在读入的时间需要进行判断

几何网格的加载比较简单，直接递归地从根结点来遍历整个graph，检测当前的结点是否为eMESH的属性，若是即处理其中的几何数据，主要代码如下所示：

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7210511#)

1. **void** ProcessMesh(KFbxNode\* pNode)
2. {
3. KFbxMesh\* pMesh = pNode->GetMesh();
4. **if**(pMesh == NULL)
5. {
6. **return**;
7. }
9. D3DXVECTOR3 vertex[3];
10. D3DXVECTOR4 color[3];
11. D3DXVECTOR3 normal[3];
12. D3DXVECTOR3 tangent[3];
13. D3DXVECTOR2 uv[3][2];
15. **int** triangleCount = pMesh->GetPolygonCount();
16. **int** vertexCounter = 0;
18. **for**(**int** i = 0 ; i < triangleCount ; ++i)
19. {
20. **for**(**int** j = 0 ; j < 3 ; j++)
21. {
22. **int** ctrlPointIndex = pMesh->GetPolygonVertex(i , j);
24. // Read the vertex
25. ReadVertex(pMesh , ctrlPointIndex , &vertex[j]);
27. // Read the color of each vertex
28. ReadColor(pMesh , ctrlPointIndex , vertexCounter , &color[j]);
30. // Read the UV of each vertex
31. **for**(**int** k = 0 ; k < 2 ; ++k)
32. {
33. ReadUV(pMesh , ctrlPointIndex , pMesh->GetTextureUVIndex(i, j) , k , &(uv[j][k]));
34. }
36. // Read the normal of each vertex
37. ReadNormal(pMesh , ctrlPointIndex , vertexCounter , &normal[j]);
39. // Read the tangent of each vertex
40. ReadTangent(pMesh , ctrlPointIndex , vertexCounter , &tangent[j]);
42. vertexCounter++;
43. }
45. // 根据读入的信息组装三角形，并以某种方式使用即可，比如存入到列表中、保存到文件等...
46. }
47. }

上述代码完成了从一个Node里边读出相应的网格信息。首先，从Node里边得到相应KFbxMesh指针，可知，如果该Node不是eMESH属性的话那么该指针就为空，后继操作不能再进行。注意其中用triangleCount变量来存储pMesh->GetPolygonCount()的值，这主要是在前面也提到过了，假定对于所有的FBX模型在存储时均选定了Triangulation的操作，因而其中存储的Polygon是三角形，如此一来每个里边一定只包含3个顶点，依次读入这3个顶点所对应的各属性信息即可。在FBX中对于每个顶点所对应的各种额外属性，比如Normal、Tangent、UV等均可对应多个通道，这可以通过在每个Mesh里边增加相应属性的一个Layer即可实现，在使用FBX SDK写出FBX文件时很容易做到。比如上述代码中就从FBX中读出4个UV通道中的值（第一个是正常的贴图通道，第二层是LightMap的通道）。vertexCounter是记录已经处理过的顶点的数目，这主要是顶点信息读取在某些映射模式下（比如下述使用到vertexCounter的eBY\_POLYGON\_VERTEX等）需要知道其在全局顶ControlPoints中的信息，因而增加这样的一个变量来进行记录。

读入顶点：

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7210511#)

1. **void** ReadVertex(KFbxMesh\* pMesh ,**int** ctrlPointIndex , D3DXVECTOR3\* pVertex)
2. {
3. KFbxVector4\* pCtrlPoint = pMesh->GetControlPoints();
5. pVertex->x = pCtrlPoint[ctrlPointIndex].GetAt(0);
6. pVertex->y = pCtrlPoint[ctrlPointIndex].GetAt(1);
7. pVertex->z = pCtrlPoint[ctrlPointIndex].GetAt(2);
8. }

读入Color:

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7210511#)

1. **void** ReadColor(KFbxMesh\* pMesh ,**int** ctrlPointIndex ,**int** vertexCounter , D3DXVECTOR4\* pColor)
2. {
3. **if**(pMesh->GetElementVertexColorCount < 1)
4. {
5. **return**;
6. }
8. KFbxGeometryElementVertexColor\* pVertexColor = pMesh->GetElementVertexColor(0);
9. **switch**(pVertexColor->GetMappingMode())
10. {
11. **case** KFbxGeometryElement::eBY\_CONTROL\_POINT:
12. {
13. **switch**(pVertexColor->GetReferenceMode())
14. {
15. **case** KFbxGeometryElement::eDIRECT:
16. {
17. pColor->x = pVertexColor->GetDirectArray().GetAt(ctrlPointIndex).mRed;
18. pColor->y = pVertexColor->GetDirectArray().GetAt(ctrlPointIndex).mGreen;
19. pColor->z = pVertexColor->GetDirectArray().GetAt(ctrlPointIndex).mBlue;
20. pColor->w = pVertexColor->GetDirectArray().GetAt(ctrlPointIndex).mAlpha;
21. }
22. **break**;
24. **case** KFbxGeometryElement::eINDEX\_TO\_DIRECT:
25. {
26. **int** id = pVertexColor->GetIndexArray().GetAt(ctrlPointIndex);
27. pColor->x = pVertexColor->GetDirectArray().GetAt(id).mRed;
28. pColor->y = pVertexColor->GetDirectArray().GetAt(id).mGreen;
29. pColor->z = pVertexColor->GetDirectArray().GetAt(id).mBlue;
30. pColor->w = pVertexColor->GetDirectArray().GetAt(id).mAlpha;
31. }
32. **break**;
34. **default**:
35. **break**;
36. }
37. }
38. **break**;
40. **case** KFbxGeometryElement::eBY\_POLYGON\_VERTEX:
41. {
42. **switch** (pVertexColor->GetReferenceMode())
43. {
44. **case** KFbxGeometryElement::eDIRECT:
45. {
46. pColor->x = pVertexColor->GetDirectArray().GetAt(vertexCounter).mRed;
47. pColor->y = pVertexColor->GetDirectArray().GetAt(vertexCounter).mGreen;
48. pColor->z = pVertexColor->GetDirectArray().GetAt(vertexCounter).mBlue;
49. pColor->w = pVertexColor->GetDirectArray().GetAt(vertexCounter).mAlpha;
50. }
51. **break**;
52. **case** KFbxGeometryElement::eINDEX\_TO\_DIRECT:
53. {
54. **int** id = pVertexColor->GetIndexArray().GetAt(vertexCounter);
55. pColor->x = pVertexColor->GetDirectArray().GetAt(id).mRed;
56. pColor->y = pVertexColor->GetDirectArray().GetAt(id).mGreen;
57. pColor->z = pVertexColor->GetDirectArray().GetAt(id).mBlue;
58. pColor->w = pVertexColor->GetDirectArray().GetAt(id).mAlpha;
59. }
60. **break**;
61. **default**:
62. **break**;
63. }
64. }
65. **break**;
66. }
67. }

读入UV:

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7210511#)

1. **void** ReadUV(KFbxMesh\* pMesh ,**int** ctrlPointIndex ,**int** textureUVIndex ,**int** uvLayer , D3DXVECTOR2\* pUV)
2. {
3. **if**(uvLayer >= 2 || pMesh->GetElementUVCount() <= uvLayer)
4. {
5. **return** **false**;
6. }
8. KFbxGeometryElementUV\* pVertexUV = pMesh->GetElementUV(uvLayer);
10. **switch**(pVertexUV->GetMappingMode())
11. {
12. **case** KFbxGeometryElement::eBY\_CONTROL\_POINT:
13. {
14. **switch**(pVertexUV->GetReferenceMode())
15. {
16. **case** KFbxGeometryElement::eDIRECT:
17. {
18. pUV->x = pVertexUV->GetDirectArray().GetAt(ctrlPointIndex).GetAt(0);
19. pUV->y = pVertexUV->GetDirectArray().GetAt(ctrlPointIndex).GetAt(1);
20. }
21. **break**;
23. **case** KFbxGeometryElement::eINDEX\_TO\_DIRECT:
24. {
25. **int** id = pVertexUV->GetIndexArray().GetAt(ctrlPointIndex);
26. pUV->x = pVertexUV->GetDirectArray().GetAt(id).GetAt(0);
27. pUV->y = pVertexUV->GetDirectArray().GetAt(id).GetAt(1);
28. }
29. **break**;
31. **default**:
32. **break**;
33. }
34. }
35. **break**;
37. **case** KFbxGeometryElement::eBY\_POLYGON\_VERTEX:
38. {
39. **switch** (pVertexUV->GetReferenceMode())
40. {
41. **case** KFbxGeometryElement::eDIRECT:
42. **case** KFbxGeometryElement::eINDEX\_TO\_DIRECT:
43. {
44. pUV->x = pVertexUV->GetDirectArray().GetAt(textureUVIndex).GetAt(0);
45. pUV->y = pVertexUV->GetDirectArray().GetAt(textureUVIndex).GetAt(1);
46. }
47. **break**;
49. **default**:
50. **break**;
51. }
52. }
53. **break**;
54. }
55. }

读入Normal:

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7210511#)

1. **void** ReadNormal(KFbxMesh\* pMesh ,**int** ctrlPointIndex ,**int** vertexCounter , D3DXVECTOR3\* pNormal)
2. {
3. **if**(pMesh->GetElementNormalCount() < 1)
4. {
5. **return**;
6. }
8. KFbxGeometryElementNormal\* leNormal = pMesh->GetElementNormal(0);
9. **switch**(leNormal->GetMappingMode())
10. {
11. **case** KFbxGeometryElement::eBY\_CONTROL\_POINT:
12. {
13. **switch**(leNormal->GetReferenceMode())
14. {
15. **case** KFbxGeometryElement::eDIRECT:
16. {
17. pNormal->x = leNormal->GetDirectArray().GetAt(ctrlPointIndex).GetAt(0);
18. pNormal->y = leNormal->GetDirectArray().GetAt(ctrlPointIndex).GetAt(1);
19. pNormal->z = leNormal->GetDirectArray().GetAt(ctrlPointIndex).GetAt(2);
20. }
21. **break**;
23. **case** KFbxGeometryElement::eINDEX\_TO\_DIRECT:
24. {
25. **int** id = leNormal->GetIndexArray().GetAt(ctrlPointIndex);
26. pNormal->x = leNormal->GetDirectArray().GetAt(id).GetAt(0);
27. pNormal->y = leNormal->GetDirectArray().GetAt(id).GetAt(1);
28. pNormal->z = leNormal->GetDirectArray().GetAt(id).GetAt(2);
29. }
30. **break**;
32. **default**:
33. **break**;
34. }
35. }
36. **break**;
38. **case** KFbxGeometryElement::eBY\_POLYGON\_VERTEX:
39. {
40. **switch**(leNormal->GetReferenceMode())
41. {
42. **case** KFbxGeometryElement::eDIRECT:
43. {
44. pNormal->x = leNormal->GetDirectArray().GetAt(vertexCounter).GetAt(0);
45. pNormal->y = leNormal->GetDirectArray().GetAt(vertexCounter).GetAt(1);
46. pNormal->z = leNormal->GetDirectArray().GetAt(vertexCounter).GetAt(2);
47. }
48. **break**;
50. **case** KFbxGeometryElement::eINDEX\_TO\_DIRECT:
51. {
52. **int** id = leNormal->GetIndexArray().GetAt(vertexCounter);
53. pNormal->x = leNormal->GetDirectArray().GetAt(id).GetAt(0);
54. pNormal->y = leNormal->GetDirectArray().GetAt(id).GetAt(1);
55. pNormal->z = leNormal->GetDirectArray().GetAt(id).GetAt(2);
56. }
57. **break**;
59. **default**:
60. **break**;
61. }
62. }
63. **break**;
64. }
65. }

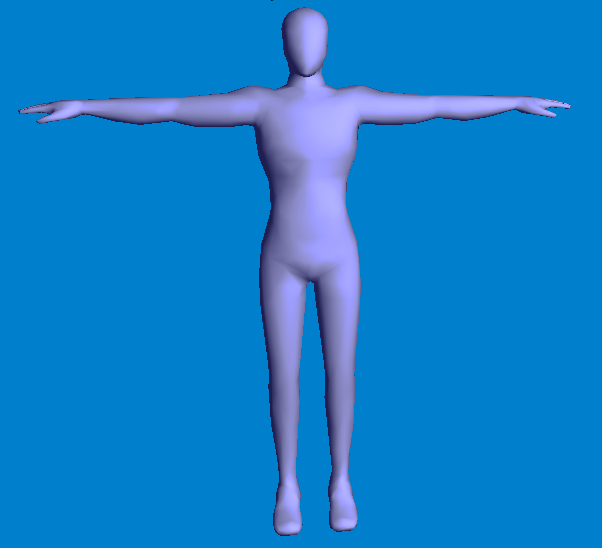
读入Tangent:

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7210511#)

1. **void** ReadTangent(KFbxMesh\* pMesh ,**int** ctrlPointIndex ,**int** vertecCounter , D3DXVECTOR3\* pTangent)
2. {
3. **if**(pMesh->GetElementTangentCount() < 1)
4. {
5. **return**;
6. }
8. KFbxGeometryElementTangent\* leTangent = pMesh->GetElementTangent(0);
10. **switch**(leTangent->GetMappingMode())
11. {
12. **case** KFbxGeometryElement::eBY\_CONTROL\_POINT:
13. {
14. **switch**(leTangent->GetReferenceMode())
15. {
16. **case** KFbxGeometryElement::eDIRECT:
17. {
18. pTangent->x = leTangent->GetDirectArray().GetAt(ctrlPointIndex).GetAt(0);
19. pTangent->y = leTangent->GetDirectArray().GetAt(ctrlPointIndex).GetAt(1);
20. pTangent->z = leTangent->GetDirectArray().GetAt(ctrlPointIndex).GetAt(2);
21. }
22. **break**;
24. **case** KFbxGeometryElement::eINDEX\_TO\_DIRECT:
25. {
26. **int** id = leTangent->GetIndexArray().GetAt(ctrlPointIndex);
27. pTangent->x = leTangent->GetDirectArray().GetAt(id).GetAt(0);
28. pTangent->y = leTangent->GetDirectArray().GetAt(id).GetAt(1);
29. pTangent->z = leTangent->GetDirectArray().GetAt(id).GetAt(2);
30. }
31. **break**;
33. **default**:
34. **break**;
35. }
36. }
37. **break**;
39. **case** KFbxGeometryElement::eBY\_POLYGON\_VERTEX:
40. {
41. **switch**(leTangent->GetReferenceMode())
42. {
43. **case** KFbxGeometryElement::eDIRECT:
44. {
45. pTangent->x = leTangent->GetDirectArray().GetAt(vertecCounter).GetAt(0);
46. pTangent->y = leTangent->GetDirectArray().GetAt(vertecCounter).GetAt(1);
47. pTangent->z = leTangent->GetDirectArray().GetAt(vertecCounter).GetAt(2);
48. }
49. **break**;
51. **case** KFbxGeometryElement::eINDEX\_TO\_DIRECT:
52. {
53. **int** id = leTangent->GetIndexArray().GetAt(vertecCounter);
54. pTangent->x = leTangent->GetDirectArray().GetAt(id).GetAt(0);
55. pTangent->y = leTangent->GetDirectArray().GetAt(id).GetAt(1);
56. pTangent->z = leTangent->GetDirectArray().GetAt(id).GetAt(2);
57. }
58. **break**;
60. **default**:
61. **break**;
62. }
63. }
64. **break**;
65. }
66. }

上述几个Normal、Tangent、UV等信息读取的函数的实现其实都差不多，首先需要判断有没有相应的Layer关联在当前的Mesh中，若有则获取其地址，然后根据不同的映射方式使用不同的方法从内存中读取相应的值即可。

完成了这些基本几何信息的读取之后即可以使用其进行渲染了：



5. 加载材质

Material是一个模型渲染时必不可少的部分，当然，这些信息也被存到了FBX之中（甚至各种贴图等也可以直接内嵌到FBX内部），就需要从FBX中加载这些信息以完成带有材质的渲染。材质的加载可以与Mesh的加载相结合来完成，但更好的方法是独立进行，这样各模块间的关系更清晰，但这就需要一个额外的操作，那就是关联Mesh与Material。FBX中的材质对象包含了丰富的信息，比如最常规的从Max中可以看到那些材质属性，如ambient、diffuse、specular的color和texture；shininess、opacity值等，更高级一点的属性诸如Effect的参数、源文件等都可以保存。它是尽可能保证从建模工具中导出时不丢失地保存材质信息，但我们在使用时却可以有选择地读取。

5.1 关联Mesh与材质

对于Material与Mesh独立加载的系统而言，首先需要读取相关的信息将两者关联起来，这些信息其实对也都存储在KFbxMesh之内（属于几何信息的一部分吧）。每个带有材质的Mesh结点上都会包含有一个类型为KFbxGeometryElementMaterial的结点（若不含有材质则该结点为空），该结点中记录了Mesh中的多边形（这里全部为三角形）与每个材质的对应关系，读取该结点中的信息建立Mesh与Material之间的连接关系，代码如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7211515#)

1. **void** ConnectMaterialToMesh(KFbxMesh\* pMesh ,**int** triangleCount ,**int**\* pTriangleMtlIndex)
2. {
3. // Get the material index list of current mesh
4. KFbxLayerElementArrayTemplate<**int**>\* pMaterialIndices;
5. KFbxGeometryElement::EMappingMode   materialMappingMode = KFbxGeometryElement::eNONE;
7. **if**(pMesh->GetElementMaterial())
8. {
9. pMaterialIndices    = &pMesh->GetElementMaterial()->GetIndexArray();
10. materialMappingMode = pMesh->GetElementMaterial()->GetMappingMode();
11. **if**(pMaterialIndices)
12. {
13. **switch**(materialMappingMode)
14. {
15. **case** KFbxGeometryElement::eBY\_POLYGON:
16. {
17. **if**(pMaterialIndices->GetCount() == triangleCount)
18. {
19. **for**(**int** triangleIndex = 0 ; triangleIndex < triangleCount ; ++triangleIndex)
20. {
21. **int** materialIndex = pMaterialIndices->GetAt(triangleIndex);
23. pTriangleMtlIndex[triangleIndex] = materialIndex;
24. }
25. }
26. }
27. **break**;
29. **case** KFbxGeometryElement::eALL\_SAME:
30. {
31. **int** lMaterialIndex = pMaterialIndices->GetAt(0);
33. **for**(**int** triangleIndex = 0 ; triangleIndex < triangleCount ; ++triangleIndex)
34. {
35. **int** materialIndex = pMaterialIndices->GetAt(triangleIndex);
37. pTriangleMtlIndex[triangleIndex] = materialIndex;
38. }
39. }
40. }
41. }
42. }
43. }

其中上triangleCount即为从pMesh中读取得到的三角形的数量，pTriangleMtlIndex是一个长度为triangleCount的数组，主要用来存储读取到的三角形对应的材质索引。注意：这里考虑的情况是对于一个三角形只对应一个材质，而一般情况下也是这样（如果是对应多个材质的话需要些许修改此处的代码）。完成Mesh的索引读取之后即可以将pTriangleMtlIndex中的值以合适的方式转储到对应的三角形列表中（或以其它的方式对应）以便在渲染时使用。

5.2 普通材质

FBX中实际存储材质信息的位置是每个Mesh中对应的一个类型为KFbxSurfaceMaterial的结点，其里边存储了普通材质的典型信息，主要包括以下属性（有一些没有列出）：

* ShadingModel                 材质的光照模型，一般为两种典型的局部光照模型：Phong、Lambert
* Emissive                          Emissive属性
* EmissiveFactor
* Ambient                           Ambient属性
* AmbientFactor
* Diffuse                             Diffuse属性
* DiffuseFactor
* Specular                           Specular属性
* SpecularFactor
* Shininess                         Sepcular的Shininess属性
* Bump                               Normal Map相关的属性
* NormalMap
* BumpFactor
* TransparentColor             Transparent属性
* TransparencyFactor
* Reflection                        Reflection属性
* ReflectionFactor

当然，在实际应用中这些属性并不一定需要全部读取，可以根据情况选择读取即可。材质的读取代码如下所述（简略版）：

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7211515#)

1. **void** LoadMaterial(KFbxMesh\* pMesh)
2. {
3. **int** materialCount;
4. KFbxNode\* pNode;
6. **if**(pMesh && pMesh->GetNode())
7. {
8. pNode         = pMesh->GetNode();
9. materialCount = pNode->GetMaterialCount();
10. }
12. **if**(materialCount > 0)
13. {
14. **for**(**int** materialIndex = 0 ; materialIndex < materialCount ; materialIndex++)
15. {
16. KFbxSurfaceMaterial\* pSurfaceMaterial = pNode->GetMaterial(materialIndex);
18. LoadMaterialAttribute(pSurfaceMaterial);
19. }
20. }
21. }

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7211515#)

1. **void** LoadMaterialAttribute(KFbxSurfaceMaterial\* pSurfaceMaterial)
2. {
3. // Get the name of material
4. pSurfaceMaterial->GetName();
6. // Phong material
7. **if**(pSurfaceMaterial->GetClassId().Is(KFbxSurfacePhong::ClassId))
8. {
9. // Ambient Color
10. fbxDouble3 = ((KFbxSurfacePhong\*)pSurfaceMaterial)->Ambient;
11. // ...
13. // Diffuse Color
14. fbxDouble3 =((KFbxSurfacePhong\*)pSurfaceMaterial)->Diffuse;
15. // ...
17. // Specular Color
18. fbxDouble3 =((KFbxSurfacePhong\*)pSurfaceMaterial)->Specular;
19. // ...
21. // Emissive Color
22. fbxDouble3 =((KFbxSurfacePhong\*)pSurfaceMaterial)->Emissive;
23. // ...
25. // Opacity
26. fbxDouble1 =((KFbxSurfacePhong\*)pSurfaceMaterial)->TransparencyFactor;
27. // ...
29. // Shininess
30. fbxDouble1 =((KFbxSurfacePhong\*)pSurfaceMaterial)->Shininess;
31. // ...
33. // Reflectivity
34. fbxDouble1 =((KFbxSurfacePhong\*)pSurfaceMaterial)->ReflectionFactor;
35. // ...
36. **return**;
37. }
39. // Lambert material
40. **if**(pSurfaceMaterial->GetClassId().Is(KFbxSurfaceLambert::ClassId))
41. {
43. // Ambient Color
44. fbxDouble3=((KFbxSurfaceLambert\*)pSurfaceMaterial)->Ambient;
45. // ...
47. // Diffuse Color
48. fbxDouble3 =((KFbxSurfaceLambert\*)pSurfaceMaterial)->Diffuse;
49. // ...
51. // Emissive Color
52. fbxDouble3 =((KFbxSurfaceLambert\*)pSurfaceMaterial)->Emissive;
53. // ...
55. // Opacity
56. fbxDouble1 =((KFbxSurfaceLambert\*)pSurfaceMaterial)->TransparencyFactor;
57. // ...
58. **return**;
59. }
60. }

上述代码就可以完成对普通属性加载。另外，材质中关联的Texture也需要进行加载，这个操作一般与一个纹理管理器结合起来进行，以便对所有的Texture与Material之间形成合理的关联，这一步的操作一般如下代码所述：

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7211515#)

1. **void** LoadMaterialTexture(KFbxSurfaceMaterial\* pSurfaceMaterial)
2. {
3. **int** textureLayerIndex;
4. KFbxProperty pProperty;
5. **int** texID;
6. MaterialTextureDesc::MtlTexTypeEnum texType;
8. **for**(textureLayerIndex = 0 ; textureLayerIndex < KFbxLayerElement::LAYERELEMENT\_TYPE\_TEXTURE\_COUNT ; ++textureLayerIndex)
9. {
10. pProperty = pSurfaceMaterial->FindProperty(KFbxLayerElement::TEXTURE\_CHANNEL\_NAMES[textureLayerIndex]);
11. **if**(pProperty.IsValid())
12. {
13. **int** textureCount = pProperty.GetSrcObjectCount(KFbxTexture::ClassId);
15. **for**(**int** j = 0 ; j < textureCount ; ++j)
16. {
17. KFbxTexture\* pTexture = KFbxCast<KFbxTexture>(pProperty.GetSrcObject(KFbxTexture::ClassId,j));
18. **if**(pTexture)
19. {
20. // Use pTexture to load the attribute of current texture...
21. }
22. }
23. }
24. }
25. }

5.3 硬件相关的材质与Effect

有过建模经验的童鞋都知道，在3D Max或Maya中可以为某些材质指定特定的Shader来完成特定的效果，这些模型在保存时也会保存相应的硬件相关的Shader到FBX模型中，因而针对这样属性的材质也需要特别的代码来进行加载。FBX里边支持嵌入CG、HLSL、GLSL等主流着色语言，而着色语言的类型在解析时也很容易得到。

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7211515#)

1. **void** LoadMaterialAttribute(KFbxSurfaceMaterial\* pSurfaceMaterial)
2. {
3. KFbxImplementation\* pImplementation;
4. KString implemenationType;
6. pImplementation = GetImplementation(pSurfaceMaterial , ImplementationHLSL);
8. KString implemenationType = "HLSL";
10. **if**(pImplementation)
11. {
12. LoadMaterialEffect(pSurfaceMaterial , pImplementation , &implemenationType);
13. }
14. }

上述代码可以与前面的Material属性读取的代码合并。FBX一般通过一个类型为KFbxImplementation的对象将硬件相关的Shader与Material进行关联，可以使用如上的代码实现两者之间关联的情况的获取，其中ImplementationHLSL为一个标识HLSL类型Shader的宏，若是CG则用ImplementationCGFX。如果当前Material中包含了HLSL类型Shader之后，那么就可以得到一个不为空的KFbxImplementation类型的指针，在其中就可以解析该Shader的属性，否则，则该指针为空，说明些材质关联了其它类似的Shader或是不包含Shader。通过KFbxImplementation来获取Effect对应的属性的代码如下所示：

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7211515#)

1. **void** LoadMaterialEffect(KFbxSurfaceMaterial\* pSurfaceMaterial ,**const** KFbxImplementation\* pImplementation , KString\* pImplemenationType)
2. {
3. KFbxBindingTable **const**\* lRootTable = pImplementation->GetRootTable();
4. fbxString lFileName                = lRootTable->DescAbsoluteURL.Get();
5. fbxString lTechniqueName           = lRootTable->DescTAG.Get();
7. // Name of the effect file
8. lFileName.Buffer();
10. KFbxBindingTable **const**\* pBTable = pImplementation->GetRootTable();
11. **size\_t** entryCount = pBTable->GetEntryCount();
13. **for**(**size\_t** i = 0 ; i < entryCount ; ++i)
14. {
15. **const** KFbxBindingTableEntry& btEntry = pBTable->GetEntry(i);
16. **const** **char**\* pEntrySrcType = btEntry.GetEntryType(**true**);
17. KFbxProperty fbxProperty;
19. // Name of Parameter
20. btEntry.GetDestination();
22. // Semantic of Parameter
23. btEntry.GetDestination();
25. **if**(strcmp(KFbxPropertyEntryView::sEntryType , pEntrySrcType) == 0)
26. {
27. fbxProperty = pSurfaceMaterial->FindPropertyHierarchical(btEntry.GetSource());
28. **if**(!fbxProperty.IsValid())
29. {
30. fbxProperty = pSurfaceMaterial->RootProperty.FindHierarchical(btEntry.GetSource());
31. }
32. }
33. **else**
34. {
35. **if**(strcmp(KFbxConstantEntryView::sEntryType , pEntrySrcType) == 0)
36. {
37. fbxProperty = pImplementation->GetConstants().FindHierarchical(btEntry.GetSource());
38. }
39. }
41. **if**(fbxProperty.IsValid())
42. {
43. **if**(fbxProperty.GetSrcObjectCount(FBX\_TYPE(KFbxTexture)) > 0)
44. {
45. // Texture Parameter
46. **for**(**int** j = 0 ; j < fbxProperty.GetSrcObjectCount(FBX\_TYPE(KFbxFileTexture)) ; ++j)
47. {
48. KFbxFileTexture\* pFileTexture = fbxProperty.GetSrcObject(FBX\_TYPE(KFbxFileTexture) , j);
49. }
51. **for**(**int** j = 0 ; j < fbxProperty.GetSrcObjectCount(FBX\_TYPE(KFbxLayeredTexture)) ; ++j)
52. {
53. KFbxLayeredTexture\* pLayeredTexture = fbxProperty.GetSrcObject(FBX\_TYPE(KFbxLayeredTexture) , j);
54. }
56. **for**(**int** j = 0 ; j < fbxProperty.GetSrcObjectCount(FBX\_TYPE(KFbxProceduralTexture)) ; ++j)
57. {
58. KFbxProceduralTexture\* pProceduralTexture = fbxProperty.GetSrcObject(FBX\_TYPE(KFbxProceduralTexture) , j);
59. }
60. }
61. **else**
62. {
63. // Common Parameter
64. KFbxDataType dataType = fbxProperty.GetPropertyDataType();
66. // Bool value
67. **if**(DTBool == dataType)
68. {
69. **bool** boolValue = KFbxGet<**bool**>(fbxProperty);
70. }
72. // Integer value
73. **if**(DTInteger == dataType || DTEnum == dataType)
74. {
75. **int** intValue = KFbxGet<**int**>(fbxProperty);
76. }
78. // Float
79. **if**(DTFloat == dataType)
80. {
81. **float** floatValue = KFbxGet<**float**>(fbxProperty);
82. }
84. // Double
85. **if**(DTDouble == dataType)
86. {
87. **double** doubleValue = (**float**)KFbxGet<**double**>(fbxProperty);
88. }
90. // Double2
91. **if**(DTDouble2 == dataType)
92. {
93. fbxDouble2 lDouble2 = KFbxGet<fbxDouble2>(fbxProperty);
94. D3DXVECTOR2 double2Value = D3DXVECTOR2((**float**)lDouble2[0] , (**float**)lDouble2[1]);
95. }
97. // Double3
98. **if**(DTDouble3  == dataType || DTVector3D == dataType || DTColor3 == dataType)
99. {
100. fbxDouble3 lDouble3 = KFbxGet<fbxDouble3>(fbxProperty);
101. D3DXVECTOR3 double3Value = D3DXVECTOR3((**float**)lDouble3[0] , (**float**)lDouble3[1] , (**float**)lDouble3[2]);
102. }
104. // Double4
105. **if**(DTDouble4  == dataType || DTVector4D == dataType || DTColor4 == dataType)
106. {
107. fbxDouble4 lDouble4 = KFbxGet<fbxDouble4>(fbxProperty);
108. D3DXVECTOR4 double4Value = D3DXVECTOR4((**float**)lDouble4[0] , (**float**)lDouble4[1] , (**float**)lDouble4[2] , (**float**)lDouble4[3]);
109. }
111. // Double4x4
112. **if**(DTDouble44 == dataType)
113. {
114. fbxDouble44 lDouble44 = KFbxGet<fbxDouble44>(fbxProperty);
116. D3DXMATRIX double4x4Value;
118. **for**(**int** i = 0 ; i < 4 ; ++i)
119. {
120. **for**(**int** j = 0 ; j < 4 ; ++j)
121. {
122. double4x4Value.m[i][j] = (**float**)lDouble44[i][j];
123. }
124. }
125. }
127. // String
128. **if**(DTString == dataType || DTUrl == dataType || DTXRefUrl == dataType)
129. {
130. **char**\* pStringBuffer =(KFbxGet<fbxString>(fbxProperty)).Buffer();
131. }
132. }
133. }
134. }
135. }

可以解析到的Effect的主要属性包括Shader所对应的源文件、Shader中提供的各种外部参数的初始设定等（比如在3D Max中通过UI控件所调节的参数的数值）。具体的方法代码里边已经比较明确了，这里就不在赘述了。后续的一些操作就要看整个材质与Effect部分的[数据结构](http://lib.csdn.net/base/31)如何组织以及如何与你自己的代码整合。

5.4 根据材质优化Mesh

通过FBX导出之后得到的FBX模型在存储时一般会以几何属性为首要考量因素来生成整个文件的Scene graph，因此上述解析得到的几何网格与Material之间的映射关系可能并不适合于直接进行绘制，一般需要重新再组织。比如其间的映射关系可能是

1. Triangle0 -> Material1
2. Triangle1 -> Material0
3. Triangle2 -> Material1
4. ...

如果一个应用的渲染流程使用了Material之间的最少切换次数来作为渲染的首要考虑的话，那么就不能直接 使用Triangle的顺序来生成渲染Buffer，而需要根据Material对其进行再排序并重新组织几何数据间的次序。

完成上述加载之后即可实现带有材质的渲染效果：



6. 加载Camera和Light

在FBX模型中除了几何数据外较为常用的信息可能就是Camera和Light，虽然在游戏中一般不直接从模型中得到这两部分信息，而是由引擎来提供，但是FBX中提供了对这些信息保存的支持。其实单纯加载这两部分的信息很简单，就像之前介绍的在整个Scene Graph中对每个Node遍历过程中，判断得到当前结点是Camera或Light时调用相应的ProcessCamera或ProcessLight来完成相关的处理操作即可。

如果对于当前结点判断得到其是一个Camera结点，那么可以直接调用GetCamera来得到一个KFbxCamera类型的指针，之后就可以通过该指针来完成Camera属性的获取。

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7229416#)

1. **void** ProcessCamera(KFbxNode\* pNode)
2. {
3. KFbxCamera\* pCamera = pNode->GetCamera();
5. // 调用相应的接口获取Camera的属性即可
6. }

对于Light结点的处理与Camera类似。至于Camera与Light结点所具有的属性可以直接在SDK中看kfbxcamera与kfbxlight的类型定义即可。

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7229416#)

1. **void** ProcessLight(KFbxNode\* pNode)
2. {
3. KFbxLight\* pLight = pNode->GetLight();
5. // 调用相应的接口获取Light的属性即可
7. }

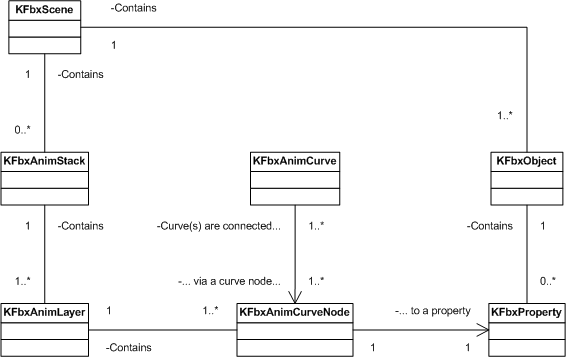
7. 加载动画

动画信息是模型数据中非常重要的一部分，也是一个渲染或游戏引擎最基本的需求之一。FBX对Animation的良好支持也成为其与.obj等静态模型最主要区别之一，而且最新的SDK中也提供了对Animation很丰富与简便的操作接口，包括自定义写入与读出等。接下来介绍一下如何使用FBX SDK来加载FBX中存储的动画信息。

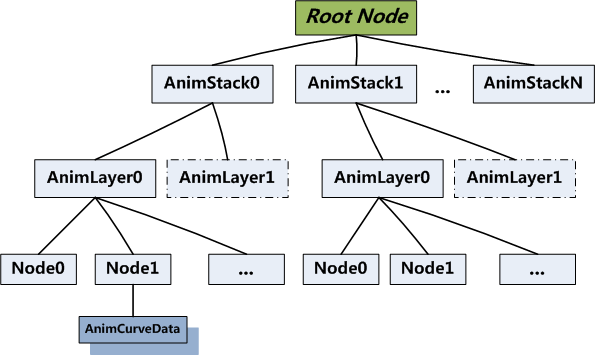
7.1 动画数据读取

在FBX中实现对于动画数据的存储主要通过以下三个对象层来实现：Animaiton Stack、 Animation Layer、Animation Node，其层次关系为

Animation Stack -> Animation Layer -> Animation Node，图示化结构为（图片来自于FBX SDKRef）：



其中的Animation Stack为FBX动画管理的最高层，其中包含着与之相关联的Animation Layer等；每个Animation Stack对应着一套动作过程。每个Stack中包含一个或多个Animation Layer(当用来做blend时就需要多个Layer，但一般是一个)。在每个Layer中又通过一个KFbxAnimCurveNode的结点使Layer与具体的动画数据发生关系。一般情况下可以根据自己的需要情况或引擎的动画实现方式来读取FBX中的动画数据，例如本人在实现时从FBX中读取数据的方法就可以抽像化为如下图所示的结构：



其中对每个Node判断其是否有对应的动画数据，若有则读取其Curve中的数据并存储以供渲染更新使用，代码如下所述：

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7229416#)

1. **void** LoadNodeCurve(KFbxAnimLayer\* pAnimationLayer , KFbxNode\* pNode , StackTimeSpan& timeSpan)
2. {
3. KTime       keyTimer;
5. unsigned **long** millseconds;
7. **for**(**UINT** i = 0 ; i < timeSpan.mKeyNums ; ++i)
8. {
9. millseconds = timeSpan.mStart + (**float**)i \* timeSpan.mStep;
10. keyTimer.SetMilliSeconds(millseconds);
12. // 计算得到当前结点在当前时刻下所对应的空间局部和全局矩阵
13. // 局部矩阵对于Skeleton是必需的，因需要使用它来计算父子Skeleton之间的空间关系
14. KFbxXMatrix curveKeyLocalMatrix  = pNode->EvaluateLocalTransform(keyTimer);
15. KFbxXMatrix curveKeyGlobalMatrix = pNode->EvaluateGlobalTransform(keyTimer);
16. }
17. }

代码中的timeSpan是一个自定义的结构，其中包含了整个FBX对象动画信息的相关数据，比如帧数、起始时间、帧间时差等；在读取时需将其中的信息转换为一个KTime类型的对象（keyTimer）以供FBX SDK的API使用。上述操作加载了动画数据中直接相关的空间Matrix信息，这是普通模型对象的基本动画信息。但是对于Camera或Light等对象而言，动画不仅包含着位置或空间信息的变化而且还包含着一些其它的属性变化如Camera的FOV，Light的Direction，Color等，这些信息也导出FBX时被存储到了FBX中。而这些信息的获取就是通过KFbxCurveNode来实现，其关联具体的Curve到相应的Property上，进而从中获得对应的动画信息。比如我们熟悉的Camera实现中有一个常用的属性PixelAspectRatio，用来描述视口Width与Height之间的比值，对于某些动画效果这个Ratio可能是时变的，因而在建模时就会将该信息同样以动画的信息进行存储，现在我们想要得到这一部分动画数据。通过查看kfbxcamera.h可以发现在KFbxCamera的定义中含有

KFbxTypedProperty<fbxDouble1> PixelAspectRatio

的一个成员变量，这即是PixelAspcetRatio动画数据所存储的位置；而在ProcessCamera时已经由当前Node的指针得到了Camera对应的指针，之后该部分读取代码基本上如下所述：

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7229416#)

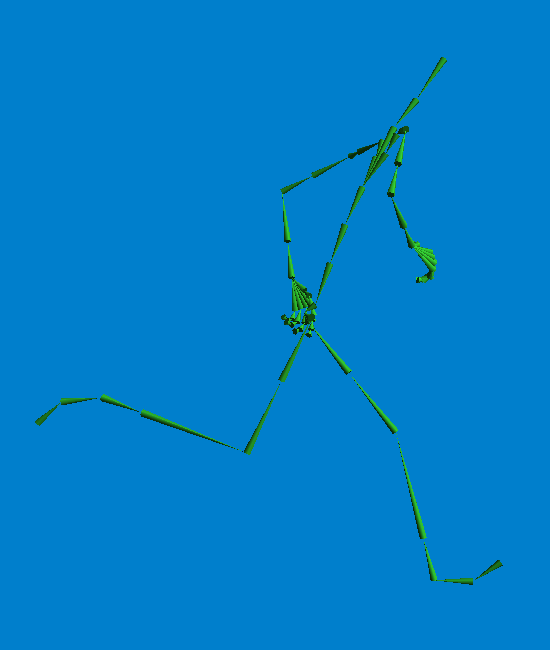
1. **void** LoadCameraCurve(KFbxAnimLayer\* pAnimationLayer , KFbxCamera\* pCamera , StackTimeSpan& timeSpan)
2. {
3. **if**(pCamera == NULL)
4. {
5. **return**;
6. }
8. // 通过FBX的属性对象而获取其所对应的Animation Curve
9. KFbxAnimCurve\* pCameraAttriAnimCurve = pCamera->PixelAspectRatio.GetCurve<KFbxAnimCurve>(pAnimationLayer);
11. // 判断当前的属性是否含有可变的Animation值
12. **if**(pCameraAttriAnimCurve)
13. {
14. KTime         keyTimer;
15. unsigned **long** millseconds;
17. **for**(**UINT** i = 0 ; i < timeSpan.mKeyNums ; ++i)
18. {
19. millseconds = timeSpan.mStart + (**float**)i \* timeSpan.mStep;
20. keyTimer.SetMilliSeconds(millseconds);
22. // 计算Camera的某属性在当前时刻所对应的值
23. pCameraAttriAnimCurve->Evaluate(keyTimer);
24. }
25. }
26. }

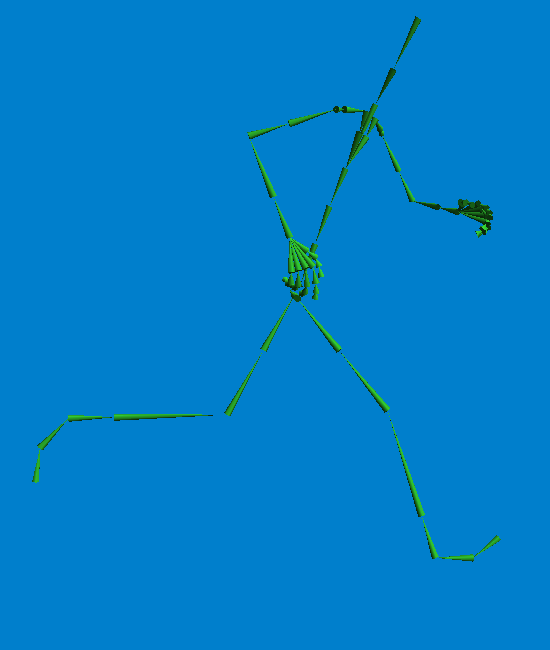
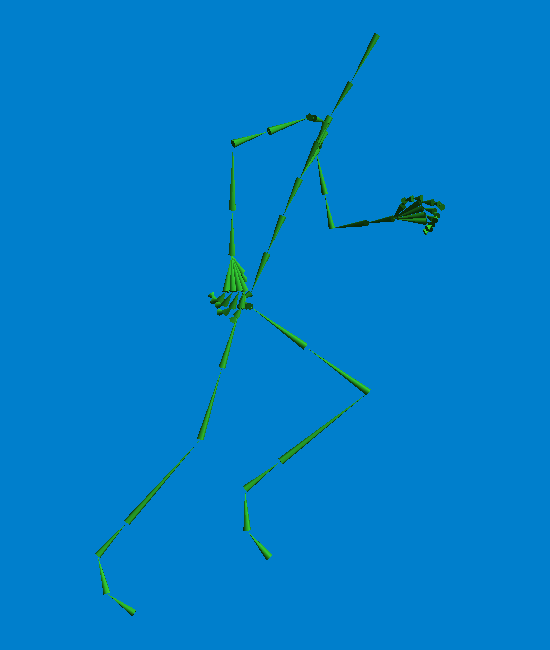
上述代码通过PixelAspectRatio的属性对象加载了其不同时刻下的动画值，其它的属性的动画读取也可以用类似的操作实现。

7.2 动画驱动

加载了上述的动画数据以后，即可以使用其来驱动模型中的直接动画相关部分，如Camera、Light、Skeleton等。由之前的代码可知，在加载动画数据时我们使用了当前Node的指针，因而就可以用它在加载动画时存储其它的一些额外信息使这些动画数据与对应的Camera、Light、Skeleton等部件进行关联（比如Node的指针，或是Node的Name等），从而可以从动画管理器中随时查得到某结点在指定时刻位置上的动画数据。该部分可以根据具体的实现采取适宜的操作即可完成。

最后，带有动画驱动的Skeleton渲染效果如下列图所示（Camera，Light的动画效果木有绘出）：

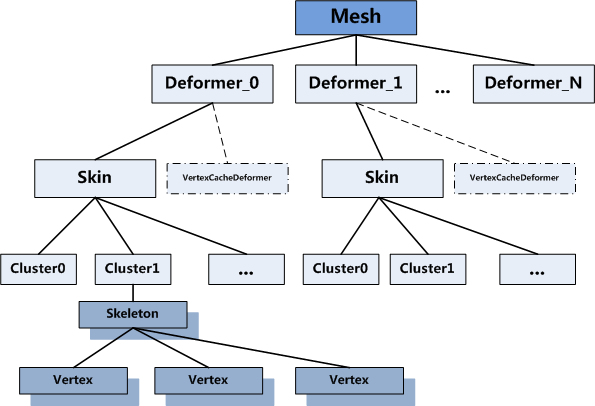




8. 骨骼蒙皮动画

骨骼蒙皮动画是当前游戏引擎中最常用的一种动画方式，关于其基本原理网络上的资料较多，关于到涉及的其它较复杂操作，如插值、融合等在这里也就先不再讨论了，而且其实现方式也与具体引擎的动作管理系统相关；在这里就主要简单介绍一下如何从FBX里加载骨骼以及蒙皮信息并完成最基本的蒙皮动画效果。骨骼动画的实现主要包括骨骼的驱动和蒙皮两部分操作，骨骼的驱动在前一篇中介绍动画数据的加载时已经完成了，接下来就是对于Mesh与Skeleton之间的Skinning操作。

我们知道，骨骼动画其实就是通过更新较少量的Skeleton，进而实现对关联到这些骨骼上的Mesh的更新，在每帧间都进行这样的更新并做合适的插值与融合就可以得到平滑流畅的动作效果了。通过前面基本几何和动画数据（Skeleton和Mesh）的加载已经有了这两部分必要信息，接下来就需要对两者进行关联从而实现Skinning时的正确映射。这一部分数据的读取其实还是以Mesh为单位进行的，其层次关系结构图如下所示：



其中的Mesh可从当前属性为eMESH的Node结点中获得（与读取几何网格数据相同），其可能是构成整个模型的网格的一小部分（Sub-Mesh）。若当前的Mesh中含有相应的蒙皮动画数据，则可以从其中读取出全部的Vertex到Skeleton的映射信息。Mesh中的蒙皮数据由一个或多个KFbxDeformer来管理，KFbxDeformer是类型为KFbxTakeNodeContainer的一个对象。每个Deformer管理当前Mesh中的部分顶点到Skeleton的映射关系，而这种映射关系的组织方式又分为两种不同的形式，因而就有了派生自Deformer的KFbxSkin和KFbxVertexCacheDeformer（一般情况下只需考虑KFbxSkin的方式）。每个Skin（Deformer）中可能对应到多个顶点，这些顶点又可能映射到多个Skeleton，在Skin（Deformer）中的每个Skeleton对应着一个Cluster。如此一来，通过在每个Cluster（->Skeleton）中寻找其所影响到的Vertex，得到相应的联接信息如映射Matrix、骨骼Weight等并做相应的存储即可完成Skeleton到Mesh之间的映射蒙皮。另外注意：Vertex和Skeleton之间的关系是多对多，即一个Vertex可能受多个Skeleton影响，而一个Skeleton又可能影响到多个Vertex；这些关系在设计数据结构时就应该有所注意。该部分的代码大体如下所述：

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7232291#)

1. **void** AssociateSkeletonWithCtrlPoint(KFbxMesh\* pMesh , CSkeletonMgr\* pSkeletonMgr , List<VertexSkeletonList>& ctrlPointSkeletonList)
2. {
3. **if**(!pMesh || !pSkeletonMgr)
4. {
5. **return**;
6. }
8. **int** ctrlPointCount = pMesh->GetControlPointsCount();
9. **int** deformerCount  = pMesh->GetDeformerCount();
11. // 初始化相应的列表
12. ctrlPointSkeletonList.SetCapacity(ctrlPointCount);
13. ctrlPointSkeletonList.setListSize(ctrlPointCount);
15. KFbxDeformer\* pFBXDeformer;
16. KFbxSkin\*     pFBXSkin;
18. **for**(**int** i = 0 ; i < deformerCount ; ++i)
19. {
20. pFBXDeformer = pMesh->GetDeformer(i);
22. **if**(pFBXDeformer == NULL)
23. {
24. **continue**;
25. }
27. // 只考虑eSKIN的管理方式
28. **if**(pFBXDeformer->GetDeformerType() != KFbxDeformer::eSKIN)
29. {
30. **continue**;
31. }
33. pFBXSkin = (KFbxSkin\*)(pFBXDeformer);
34. **if**(pFBXSkin == NULL)
35. {
36. **continue**;
37. }
39. AssociateSkeletonWithCtrlPoint(pFBXSkin , pSkeletonMgr , ctrlPointSkeletonList);
40. }
41. }

**[cpp]** [view plaincopy](http://blog.csdn.net/bugrunner/article/details/7232291#)

1. **void** AssociateSkeletonWithCtrlPoint(KFbxSkin\* pSkin , CSkeletonMgr\* pSkeletonMgr , List<VertexSkeletonList>& ctrlPointSkeletonList)
2. {
3. **if**(!pSkin || !pSkeletonMgr)
4. {
5. **return**;
6. }
8. KFbxCluster::ELinkMode linkMode = KFbxCluster::eNORMALIZE;
9. KFbxCluster\* pCluster;
10. KFbxNode\*    pLinkNode;
11. **int**          skeletonIndex;
12. CSkeleton\*   pSkeleton;
13. KFbxXMatrix  transformMatrix , transformLinkMatrix;
14. **int**          clusterCount = pSkin->GetClusterCount();
16. // 处理当前Skin中的每个Cluster（对应到Skeleton）
17. **for**(**int** i = 0 ; i < clusterCount ; ++i)
18. {
19. pCluster = pSkin->GetCluster(i);
21. **if**(!pCluster)
22. {
23. **continue**;
24. }
26. pLinkNode = pCluster->GetLink();
28. **if**(!pLinkNode)
29. {
30. **continue**;
31. }
33. // 通过Skeleton管理器搜索到当前Cluster所对应的Skeleton，并与Cluster进行关联
34. skeletonIndex = pSkeletonMgr->FindSkeleton(pLinkNode->GetName());
36. // ... //关联Skeleton与Cluster
38. **if**(skeletonIndex < 0)
39. {
40. **continue**;
41. }
43. pSkeleton = pSkeletonMgr->GetSkeleton(skeletonIndex);
45. // 得到每个Cluster（Skeleton）所对应的Transform和TransformLink矩阵，后面具体说明
46. pCluster->GetTransformMatrix(transformMatrix);
47. pCluster->GetTransformLinkMatrix(transformLinkMatrix);
49. // 其它适宜的操作，将Transform、TransformLink转换为映射矩阵并存储到相应的Skeleton中
50. // ...
52. **int**     associatedCtrlPointCount = pCluster->GetControlPointIndicesCount();
53. **int**\*    pCtrlPointIndices = pCluster->GetControlPointIndices();
54. **double**\* pCtrlPointWeights = pCluster->GetControlPointWeights();
55. **int**     ctrlPointIndex;
57. // 遍历当前Cluster所影响到的每个Vertex，并将对相应的信息做记录以便Skinning时使用
58. **for**(**int** j = 0 ; j < associatedCtrlPointCount ; ++j)
59. {
60. ctrlPointIndex = pCtrlPointIndices[j];
61. ctrlPointSkeletonList[ctrlPointIndex].AddSkeleton(skeletonIndex , pCtrlPointWeights[j]);
62. }
63. }
64. }

上述代码只是完整代码的一部分，因其中涉及的大多数操作都与具体的实现系统相关，这里只列出部分以供参考而己。其中有两个操作 pCluster->GetTransformMatrix(transformMatrix); pCluster->GetTransformLinkMatrix(transformLinkMatrix); 需要特别说明一下，两个操作分别得到两个Matrix，前者transformMatrix（记为Mt）用来描述当前映射时刻（初始的映射状态下）Mesh的变换矩阵（顶点的变换矩阵）， 后者transformLinkMatrix（记为Mtl）用来描述当前映射时刻Bone的变换矩阵（可以参考kfbxcluster.h中的说明）。假设通过当前的Cluster可以关联顶点V和骨骼B，而其对应的空间变换矩阵分别为MV、MB，因而有 MV = Mt; MB = Mtl 而在Mesh到Skeleton的蒙皮中需要由Skeleton的空间位置变换得到Mesh（顶点）的空间位置，所以就需要这样一个变换矩阵M使得

 通过简单的变换即可得到 

该M在动画更新时就可以用来做Skeleton到Mesh之间的映射计算。 然后，即可以通过Skeleton的更新而完成对Mesh的更新，进而得到对整个模型的动画。比如下列图所示的一套动作：





9. 其它一些问题

虽然FBX SDK提供了对FBX模型的很友好的操作接口，但是目前的发布版本也有一些相应的问题：

* FBX SDK提供的FBXImporter目前不支持中文路径，因而提供的fbx源文件地址中应不含有中文字符。
* 3D Max或Maya中的FBX导出插件计算得到的Tangent会一些问题，特别是在那些具有对称属性UV的部位。
* 导出的具有Smooth特性的Normal也会在某些网格接口处出现不平滑的现象。

后两个问题某些情况下的影响会比较严重，但是既然已经将原始的几何数据加载到自己的引擎中了，因而也就可以在引擎中对Tangent与Normal进行再计算。

前述内容介绍了使用FBX SDK来对FBX进行加载时涉及到的比较常见的操作，如加载网格、材质以及动画等，也给出了部分实现的代码，但毕竟不同的系统对各种资源（如Animation、Skeleton、Material等）有不同的管理方法，代码也不能完全直接使用，适宜地修改是必不可少的。而且其中的错误也是难免的，所以上述介绍内容只作为参考，具体的实现还需要好好研究与参考Autodesk的相关doc。

最后，欢迎交流与讨论~~