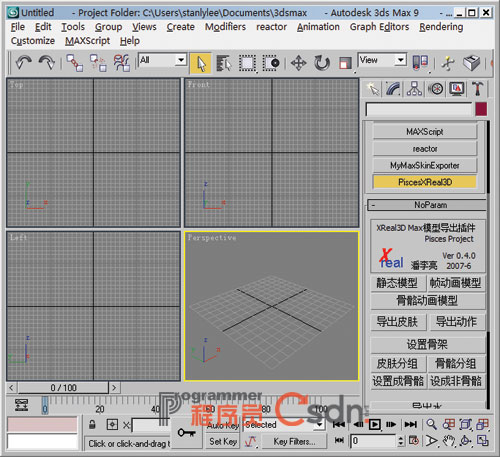
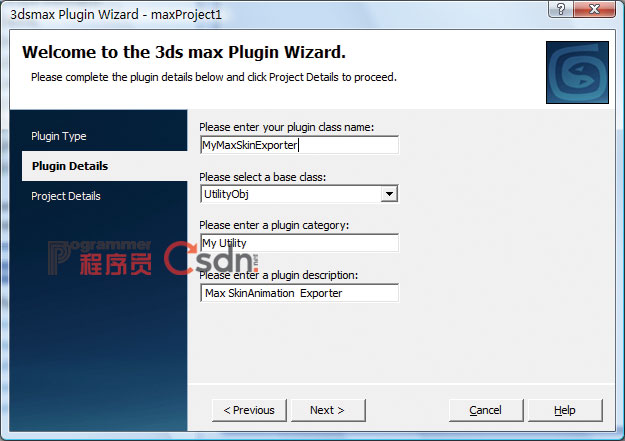
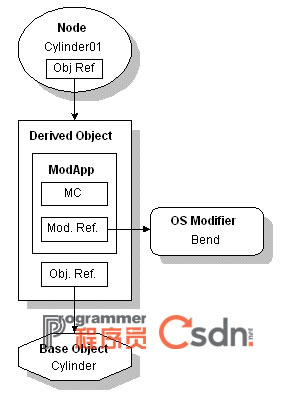
# 游戏程序中的骨骼插件

**文/潘李亮  
  
引言  
  
在3D引擎中，骨骼动画系统是非常重要的一个组成部分，虽然在一个游戏的真正开发过程中，一个优秀的游戏引擎也许不需要用户去关心它的骨骼动画系统是如何实现的，但是还是有很多人希望了解这样的一个技术。  
  
本文将会介绍骨骼动画系统里的一个基础部件：3Ds MAX 的骨骼动画导出插件。  
  
3Ds Max SDK和插件系统  
  
最新版本MAX9的MAXSDK包含在安装光盘里，在安装完MAX后直接安装SDK，并在工程里添加maxsdk的包含路径和库的路径就可以开始编译max插件了。MAX SDK还提供了3Ds Max Help for Visual Studio，这个帮助系统可以集成到Visual Studio .NET的帮助系统中，非常方便。建议在安装的时候一起装上。  
  
MaxSDK主要目的就是用来开发MAX插件，虽然Max也提供了MaxScript，也可以用来做插件，但是对C++程序原来说，MaxSDK则更顺手一些。  
  
Max插件根据用途分为好几种，每种对应不同的扩展名，在游戏开发中，我们通常比较关心三种类型的插件，他们分别是: 导入/导出插件，对应扩展名为dli/dle， utility 插件，对应扩展名为dlu，以及扩展名为dlm的modifier。导入导出插件基本上说是MAX与其它工具交互的接口。Utility插件则可以为 MAX增加很多操作功能面板。Modifier则是3DsMAX  
  
3DsMax自带的插件放在X:/3DsMax/maxsdk/stdplugs目录下，而我们自己编写的插件通常会放到X:/3DsMax /maxsdk/ plugins目录下。只要把插件放到这两个目录下，Max在启动的时候就会自动加载你的插件。很多初学者可能会问dlm/dle这些插件是怎么生成的呢？其实这些都是一些标准的dll程序，只是扩展名不同而已。跟编译一个普通的Windows DLL没有区别。  
  
  
  
初学MaxSDK最好的例子应该就是MAXSDK自带的sample。在maxsdk的安装目录下可以找到，一般是X:/3DsMax /maxsdk/samples 下。这个目录下已经对插件的种类进行了分类。一般在做骨骼动画导出插件的时候，我们不会选择导出插件而是选择utility插件，这样做的目的是 ultility插件在max启动的时候就处于激活状态，而导出插件则只会在用户选择export命令的那一刻，并且这些插件都可以访问到MAX的整个环境，因此，使用utility插件会让用户更加的方便，本文的例子就是采用utility插件。  
  
构造第一个3Ds Max 插件   
  
本节我将讲述如何快速的建立一个utility插件的框架，因为关心的是导出插件本身的功能，而不是插件框架本身，因此我给大家提供一个个比较简洁的方法:使用3dsmaxPluginWizard. 这是MaxSDK提供的一个组件，位于X:/3dsmax/maxsdk/howto/3dsmaxPluginWizard下，仔细阅读一遍这个目录下的ReadMe.txt文件的Installing一小节，就可安装好3DsMaxPluginWizard. 这时候打开Visual Studio 2005.在新建工程中就可以看到3Ds Max Plugin Wizard一项，选择后，看到标签页一共有三页，在第一页Plugin-Type里，选择Utility项，在接下来的Plugin Detail里填入详细信息如图2所示。  
  
  
  
最后在Project Detials 选项卡里填入maxsdk的路径，插件输出路径和3dsmax.exe所在的路径就可以生成一个utility工程了。  
  
生成的工程仅仅是一个架子，它包含了两个类和一个IDD\_PANEL的对话框。第一个类MyMaxSkinExporter是从 UtilityObj派生下来的，代表了插件本身。另外一个类从ClassDesc2派生，用来描述这个插件的一些信息。IDD\_PANEL则是我们插件的主界面，我们可以简单的理解它就是我们插件的主窗口。  
  
MyMaxSkinExporter有两个重要的函数: BeginEditParams(Interface \*ip，IUtil \*iu)和EndEditParams(Interface \*ip，IUtil \*iu)这两个函数。BeginEditParams可以简单的理解成插件的初始化函数，EndEditParams则在插件退出时候被调用。参数 Interface \*ip 则代表整个Max对象，用它可以访问到MAX程序的所有功能。  
  
编译这个工程，一个简单的utility插件就已经生成了，你可以在刚才Project Detials选项卡里填入的插件输出路径里找到生成的插件。  
  
3Ds MAX的场景组织和几何管道  
  
要编写一个导出骨骼动画的插件，必须先了解MAX是如何组织场景的，并了解MAX中一个mesh对象从建立到最终输出都经历那些阶段。下面首先介绍一下MAX的场景组织。  
  
MAX的整个场景是一个树状结构，树的节点用INode来表示，整个树的根节点可以通过Interface::GetRootNode来获得，场景中的所有物体都是INode。INode中的NumberOfChildren函数和GetChildNode则用来访问INode的子节点。要遍历整个场景中对象，只需要通过Interface::GetRootNode和GetChildNode做一个递归Ñ¬环就可以了。如果仅仅是想获得在视口中选定的物体，则可以使用Interface::GetSelNodes函数。  
  
INode仅仅是一个虚拟的节点，它本身仅仅包含一些标记和变换信息，并不表示实际的Object。实际的Object需要附着在INode上，并以INode的坐标系为Object的本地坐标系。Max中常见的Object有形状(各种参数曲线)，Camera，Mesh等。Object有自己的变换矩阵(TM)， 在很多情况下这个矩阵都是单位矩阵。  
  
  
  
INode的变换矩阵可以通过INode::GetNodeTM来获得，而附着在INode上的Object的变换矩阵则通过 INode::GetObjectTM来获得，因为Object相对于Node的变换矩阵通常是单位矩阵，GetNodeTM和GetObjectTM获得的矩阵通常也是一样的，但是在必要的时候一定要加以区别。关于INode和Object的变换矩阵问题的详细讨论可以参考我blog上的一篇文章：http://blog.csdn.net/Nhsoft/archive/2005/01/06/241629.aspx  
  
接下来我来看一下3DsMax一个几何物体的Pipeline。前面说过Object是附着在INode上的，在MAX里，Node有一个 Object Reference的指针，指向一个物体对象。熟悉MAX的操作方式的读者都知道，我们在MAX里建立一个对象后，可以在上面添加各种修改器 -Modifier。在Max的几何管道中，我们建立的对象通常称为Base Object。所有施加在这个物体上的修改器形成一个修改器堆栈-ModStack。BaseObject经过这个ModStack后形成一个新的 Object Reference。ModStack中的每个Modifier都是输入一个Object Reference，输出一个Object Reference， 并且在应用第一个Modifier的时候会自动在几何管道里插入一个Derived Object。最终INode的Object Reference将指向这个Derived Object。  
  
Modifier在管道中的应用实例是ModApp对象，一个ModApp代表一个应用在Object上Modifier修改，ModApp包含一个ModContext的数据对象，Modifier用ModContext中的数据来对Object进行修改，以生成最终数据。  
  
修改器按照应用的坐标系不同分成局部空间修改器和世界空间修改器(World Space Modifier)。局部空间修改器仅仅在Object的局部空间中修改Object，不会对坐标系造成影响。世界空间的修改器比如水波纹修改器则要求先将物体变换到世界空间后再进行修改，修改完成后的坐标也是世界空间的坐标。相对来说处理世界空间修改器会麻烦的多。（如果一个物体应用了世界空间修改器，则通过Mesh对象取得的坐标已经是世界坐标系中的了。不需要再乘以INode::GetObjectTM了）。  
  
导出骨骼动画数据  
  
了解了MAX的场景管理和几何管道以后，我们就可以很方便的建立一个如何取得MAX场景中定点数据的流程了。  
  
骨骼动画系统，首先应该包括物体的蒙皮数据和顶点与骨骼的绑定信息。我们分两部分介绍皮肤数据的导出。第一步，我们要导出蒙皮数据，为了简单起见，在这里只导出蒙皮的位置，法向量与切向量纹理坐标等信息留给读者自己去研究。在3DsMax里。要建立骨骼动画模型，可以使用两种修改器Skin 和，Physique。其中Physique是属于CharacterStudio的，他的API相对比较复杂，本文只介绍使用Skin修改器制作的骨骼动画模型文件。  
  
在界面上增加一些按钮  
  
在了解了那么多理论后，我们可以开始做一些实质上的事情了，首先我们要给我们的插件增加一些按钮，通过这些按钮，使用可以下达保存/加载骨架，导出动作，导出皮肤等任务。  
  
我们在第三节中生成的IDD\_PANEL的对话框中加入几个按钮，分别用于保存骨架，加载骨架，导出动作，导出皮肤。并在对话框的WM\_COMMAND消息中加入按钮响应代码。对话框的窗口过程为MyMaxSkinExporterDlgProc。  
  
增加完的IDD\_PANEL对话框看上去如图4。  
  
  
  
定义顶点数据类型  
  
骨骼动画的顶点数据应该包含顶点位置，纹理坐标，法向量，影响的骨骼编号和权重，一般影响到某个顶点的骨骼数目不会超过４个。同时，顶点位置也有两种记录方法：相对于世界空间的和相对骨骼空间的，这里我们采用相对于世界空间的记录方法，因为这种方案比较直观，只需要记录一个顶点位置就可以。麻烦的地方在于，因为骨骼的变换矩阵要求顶点是相对于该骨骼的局部空间的，因此顶点在参与骨骼蒙皮计算的时候，需要先乘上骨骼的初始位置的矩阵的逆，以变换到骨骼空间。  
  
   struct Vertex\_t  
  
   {  
  
       Point pos , normal , texCoord；  
  
      int    matID;  
  
       int   nEffBone;  
  
       struct{  
  
          int    boneIdx;  
  
           float  weight;  
  
       }Bone[4];  
  
   };  
  
导出骨架  
  
骨骼动画系统中骨架为动画的载体，所有的蒙皮都附着在骨架之上。同时要保证属于一个角色的所有的蒙皮都使用同一个骨架来建立和导出，这是一套换装系统的基本需求。因此骨架的导出和保存通常是一次性的，后续导出皮肤的时候都应该以这个骨架为基准。这也要求我们在导出骨架的时候就需要导出所有的骨骼。  
  
骨架上的骨骼其实也是一个INode，骨骼仅仅是一些变换矩阵的信息而已。目前没有特别好的办法鉴定哪些INode是骨骼，比较可行的办法是把所有Skin修改器使用到的INode都列为骨骼，同时美工还可以手动指定哪些Node为骨骼，并把这些标记用INode:: SetUserPropBool("IsBone"，bIsBone);记录到MAX文件中。  
  
保存骨架的时候，需要保存骨骼的父子关系。并需要保存这个骨骼的第一帧数据。这要求如果美工在两个不同的MAX文件里制作不同的动作的时候，除了保证骨架相同以外，第一帧也需要完全相同．  
  
骨架的保存和加载代码如下：  
  
   struct Bone\_t{  
  
    Matrix NodeInitTM;  
  
    char   Name[32]，ParantName[32];  
  
   };  
  
   class CSkeleton {  
  
   public:    vector<Bone\_t> m\_Bones;  
  
       void loadSkeleton(const char\* skeFileName){  
  
      ifstream in(skeFileName , ios::binary);  
  
       while(!in.eof()){  
  
       Bone\_t \_bone;  
  
       in.read((char\*)&\_bone , sizeof(Bone\_t));  
  
       m\_Bones.push\_back(\_bone);  
  
       }  
  
       in.close();  
  
       }  
  
   int findBoneIndex(INode\* pNode)    {//  
  
   for(int i = 0  ; i  < m\_Bones.size() ; i ++)  
  
        if(string(m\_Bones[i].Name) == pNode->GetName() )    return i;  
  
           return -1;  
  
  }  
  
    void saveBone(ostream& out , INode\* pNode , bool bRoot){  
  
       Bone\_t \_bone;  
  
       \_bone.NodeInitTM = pNode->GetNodeTM(0)  ;  
  
       strncpy(\_node.Name， pNode->GetName() , 32);  
  
       if(bRoot) \_bone.ParantName[0] = 0;  
  
       else{  
  
    INode\* pPNode =pNode->GetParantNode();  
  
      strncpy(\_bone.ParantName， pPNode->GetName() , 32);  
  
       }  
  
     out.write( (char\*)&\_bone , sizeof(Bone\_t) );  
  
     for(int i = 0 ; i < pNode->NumberOfChildren() ; i ++)  
  
     saveBone(out，pNode->GetChildNode(i)， false)；  
  
  }  
  
     void saveSkeleton(const char\* skeFileName , INode\* pRootNode){  
  
      ofstream out(skeFileName , ios::binary);  
  
      saveBone(out , pRootNode , true);  
  
      out.close();  
  
   }  
  
   };  
  
findBoneIndex函数的目的是在把从文件中加载的骨骼和MAX中的Node对应起来.因为是根据名字来进行查找比较，因此要求所有的 Node都必须要有唯一的名字．同时，骨骼之间的父子关系也是通过名字来标记的．每个Bone都记录了它的父节点的名字．Save Skeleton骨架的按钮响应代码如下:  
  
   void OnSaveSkeleton()  
  
   {  
  
          CSkeleton\* pSkeleton = GetGlobalSkeleton();  
  
              Assert(ip->GetSelNodeCount() == 1); //导出骨架的时候只能选择一个节点  
  
  
  
              const char\* filename = GetSaveFileName() ;  
  
  
  
              if(filename){  
  
              pSkeleton-> saveSkeleton(filename , ip->GetSelNode(0) );  
  
}  
  
   }  
  
导出骨架动作  
  
骨架导出后，我们需要进一步导出这个骨架的动作。在导出动作的时候，需要加载一个事先已经导出的骨架。然后遍历这个骨架中所有的骨骼，找到这个骨骼对应的INode对象。然后确定动画的长度和帧数，为每一个骨头的保存一个变换矩阵。  
  
void OnExportAnimation()  
  
   {  
  
           const char\* fileName = GetSaveFileName();  
  
ofstream out(fileName , ios::binary);  
  
Interval ARange =  ip->GetAnimRange(); //获得动画的长度  
  
  
  
            TimeValue   tAniTime =  ARange.End() - ARange.Start();  
  
        TimeValue   tTime = ARange.Start();  
  
        int nFrame = tAniTime/GetTicksPerFrame();  
  
//计算动画帧数  
  
out.write((char\*)&nFrame , sizeof(int));  
  
//记录有多少frame;  
  
                for(int i = 0 , ; i < nFrame ; i ++ ,tTime += GetTicksPerFrame()){  
  
   CSkeleton\* pSkeleton = GetGlobalSkeleton();  
  
   for(int iBone = 0 ; iBone < pSkeleton->m\_Bones.size() ; iBone ++){  
  
   Bone\_t& bone = pSkeleton->m\_Bones[iBone];  
  
   INode\* pBoneNode = GetNodeByName(bone.Name);  
  
//通过名字获得INode指针  
  
   Matrix mat = pBoneNode->GetNodeTM(tTime，NULL);  
  
                        out((char\*)&mat , sizeof(Matrix));  
  
              }  
  
   }  
  
                out.close();  
  
   }  
  
这里演示里我们记录的是骨骼的绝对变换矩阵，而不是相对父骨骼的变换矩阵，这省去了我们从根骨骼开始计算骨架的麻烦，但是也多了很多限制，比如不能进行动作混合，不能做动作的插值等，使用相对父骨骼的局部矩阵的算法留给读者自己去实现，也可以参考Cal3D和我开源的XReal3D的导出插件。此外，因为我们在顶点数据中只保存了相对世界空间的位置，所以骨骼中的NodeInitTM将用来把相对世界空间的顶点位置变换到骨骼的局部空间中，皮肤混合的时候计算公式将如下：  
  
http://images.csdn.net/20080603/02-%E6%8F%92%E4%BB%B605.jpg  
  
其中M(t，i)为第i块骨头在t时刻的变换矩阵。  
  
同样的，我们只是简单的导出每一帧的变换矩阵，而没有处理关键帧，使用关键帧加上相对父节点的局部变换矩阵的四元数插值，在保准动作的准确性前提下能大大的降低动作文件磁盘占用。**

**文/潘李亮  
  
查找Skin修改器  
  
要找到一个Mesh上是不是有Skin修改器，根据MAX的几何管道的结构，需要遍历整个ModStack中的Derived Object。判断应用在这些Derived Object上的修改器的类型。MAX中所有的对象都有一个类似COM的GUID的唯一标记ClassID。Skin修改器的ClassID为 SKIN\_CLASSID，在获得Derived Object的修改器后只需要检查修改器的ClassID是不是SKIN\_CLASSID即可。示例代码如下：  
  
ISkin \* FindSkinModifier(INode \*pINode){  
  
  Object \* pObject = pINode->GetObjectRef();  
  
  if(pObject == 0) return 0;  
  
  // 循环检测所有的DerivedObject  
  
  while(pObject->SuperClassID() == GEN\_DERIVOB\_CLASS\_ID)  
  
  {  
  
    IDerivedObject \* pDerivedObject =   
  
      static\_cast<IDerivedObject \*>(pObject);  
  
    for(int stackId = 0;   
  
      stackId < pDerivedObject->NumModifiers();   
  
      stackId++)  
  
    {  
  
      Modifier \* pModifier =   
  
        pDerivedObject->GetModifier(stackId);  
  
      //检测ClassID是不是Skin修改器  
  
      if(pModifier->ClassID() == SKIN\_CLASSID) {  
  
   return (ISkin\*) pModifier->GetInterface(I\_SKIN);  
  
      }  
  
    }  
  
    //下一//个Derived Object      
  
    pObject = pDerivedObject->GetObjRef();  
  
  }  
  
  return 0;  
  
}  
  
获取Mesh对象  
  
根据第四节中描述的，要从一个INode中获得Mesh对象，首先应该从INode中获得Object对象，然后再转成Mesh对象。具体代码如下：  
  
Mesh\* GetMesh(INode\* pNode , int iMaxTime){  
  
  NullView view;   
  
  //NullView是自定义的View类。详细参见完整的插件代码  
  
  BOOL  bNeedDelete = false;  
  
  ObjectState os = pNode->EvalWorldState(iMaxTime);  
  
  Object\* pObj =  os.obj;  
  
  TriObject \* triObject = (TriObject \*)pObj->  
  
  ConvertToType(iMaxTime, triObjectClassID);  
  
  GeomObject\* pGeoObj = (GeomObject\*)pObj;  
  
  Mesh \* pMesh = pGeoObj->GetRenderMesh(   
  
    iMaxTime , m\_pNode , view , bNeedDelete );  
  
  return pMesh;  
  
}  
  
获取皮肤数据与顶点的骨骼绑定信息  
  
在成功获取到一个节点的ISkin对象和Mesh以后，就可以使用这两个对象来提取物体的顶点数据和骨骼的绑定信息了。  
  
Mesh中的数据保存在不同的数组中，常用的包含以下几种：顶点位置信息，颜色信息，法向量，UV坐标，MapChannel信息等。其中法向量的信息不是特别的准确，需要考虑平滑组，面法向量与顶点法向量的差异等。MapChannel用于仅仅有多层纹理贴图坐标的情况，在只有一层纹理坐标的情况下则不需要考虑，使用UV坐标就足够了。本文仅仅演示如何导出顶点位置，单层纹理和骨骼绑定信息，如何准确的计算法向量以及处理多层纹理坐标的内容不在本文的讨论范围。  
  
Mesh中的numVerts变量标记了Mesh有几个顶点位置，numTexCoords标记了有几个纹理坐标。而通常这两个值是不一样的。  
  
要提取骨骼绑定信息，首先需要从ISkin对象中查询到ISkinContextData接口： ISkinContextData\* pSkinContext  
  
= pSkin->GetContextInterface(pNode); 通过ISkinContextData的GetNumAssignedBones函数可以得到影响到这个顶点的骨骼的个数，进而通过 GetAssignedBone和GetBoneWeight接口可以到这个Bone的index已经这个Bone对这个顶点的权重。  
  
void GetVertexBoneInfo(INode\* pNode , ISkin\* pSkin , Mesh\* mesh ,  int vertexIdx , int uvIdx , Vertex\_t& vOut) {  
  
  CSkeleton\* pSkeleton = GetGlobalSkeleton();  
  
  ISkinContextData\* pSkinCtx =   
  
    pSkin->GetContextInterface(pNode);  
  
  int nBones =   
  
    pSkinCtx->GetNumAssignedBones(vertexIdx);  
  
  vOut.pos = mesh->verts[vertexIdx] \*   
  
    pNode->GetObjectTM(0,NULL);//第一帧的数据  
  
  vOut.texCoord = mesh->texCoords[uvIdx];  
  
  vOut.nEffBone = nBones;  
  
  for(int jBone = 0; jBone < nBones; jBone ++) {  
  
    INode\* pBone = pSkin->GetBone(  
  
      pSkinCtx->GetAssignedBone(vertexIdx, Bone));  
  
    vOut.Bone[jBone].weight  =   
  
      pSkinCtx->GetBoneWeight(vertexIdx,jBone);  
  
    vOut.Bone[jBone].boneIdx =   
  
      pSkeleton->findBoneIndex(pBone);  
  
  }  
  
}  
  
上述示例代码中并没有考虑到有个别顶点的骨骼数量超过4种情况，在正式的代码中应该对所有影响到这个顶点的骨骼权重进行排序，取前4个权重最大的，丢弃其余的骨骼。并要考虑有有些重复骨骼的问题。  
  
导出面的和材质信息  
  
在成功解决了顶点和骨骼的数据提取后,还需要获得面的信息,即顶点的拓扑关系。3D渲染器能处理的面一般都是三角形面，因此一般要求美工在制作模型的时候首先将模型转换成Editable Mesh。这样能确保在Mesh取到的面都是三角形。  
  
MAX中的面有两种，用来表示形状的Face类和用来表示纹理的TVFace类。Face和TVFace是一一对应的。就是说要获得一个三角形的完整数据，必须同时获取Face类和TVFace类。  
  
Face类中的信息比较重要的有三个顶点的位置索引，可以通过Face::v[i]来获得，得到这个索引后，在Mesh::verts数组里就可以获得位置的数据。同理TVFace也一样。  
  
Face类里除了顶点信息外，还保存了和材质相关的信息：MaterialID。一个Mesh上通常只能应用一个Material，那为什么会有 MaterialID这个概念呢？因为在Max里除了标准材质以外，还有一种美工非常有用的材质叫多材质（Multi-Material），这种材质可以可以包含很多个标准材质，我们可以通过判断材质的ClassID来判断它是不是一个多材质，如果是，就遍历它的所有子材质（Sub-Mateiral）。 MaterialID就是对应的子材质的序号。在绘制的时候，MaterialID相同的三角形表示它们有相同的纹理和材质，在导出的时候应该按照 MaterialID进行排序。  
  
Max中的Material使用Mtl类表示，可以通过INode::GetMtl()来获得。Mtl:: NumSubMtls加GetSubMtl则可访问到这个Material的所有Material。对于一个标准材质，我们可以获得这个材质的各种属性，包括纹理贴图，纹理贴图使用的贴图坐标通道（MapChannel）,以及环境光，高光等属性。处理材质的示例代码如下：  
  
void SaveMaterial(const char\* fileMat , INode\* pNode){  
  
  ofstream out(fileMat,ios::binary);  
  
  Mtl\* pMtl = pNode->GetMtl();  
  
  int nSubMat = pMtl->NumSubMtls();  
  
  if (nSubMat == 0) {  
  
    //处理和保存一个标准材质的代码，详见参考资料  
  
    saveStdMaterial(out , (StdMat\*)pMtl);   
  
  } else {  
  
    for(int i = 0 ; i < pMtl->NumSubMtls(); i++ )  
  
      saveStdMaterial(out ,   
  
       (StdMat\*)pMtl->GetSubMtl(i));        
  
  }  
  
}  
  
   以下为简单的提取三角形数据的代码。  
  
void SaveMesh(const char\* fileMesh , INode\* pNode , Mesh\* pMesh) {  
  
  ISkin\* pSkin = GetSkinModifier(pNode);  
  
  ofstream out(fileMesh , ios::binary);  
  
  int nFace = pMesh->numFaces ;    
  
  out.write( (char\*)&nFace,sizeof(int) );  
  
  for(int i = 0 ;i < pMesh->numFaces ; i ++) {  
  
    TVFace& tvface = pMesh->tvFace[i];  
  
    Face& face = pMesh->faces[i];  
  
    for(int j = 0 ; j < 3 ; j++) {  
  
      //一个三角形三个顶点  
  
       Vertex\_t vert ;  
  
       vert.matID = face.getMatID();  
  
       GetVertexBoneInfo(pNode, pSkin, pMesh,  
  
         face.v[j], tvface.getTVert(j), vert);  
  
       //保存到文件  
  
       out.write( (char\*)&vert,sizeof(vert) );   
  
    }  
  
  }  
  
}  
  
   上述代码仅仅是简单的保存了每一个面的信息，真正应用的时候，至少还应该把所有的面按照MateiralID来进行排序，并且 GetVertexBoneInfo函数在生成顶点数据的时候必然有很多顶点是完全相同的，应该把这些相同的顶点都去掉。面的信息使用Index Buffer来保存。鉴于处理这些代码过长，就不在文中一一举例。  
  
除了顶点材质相关的信息之外，Face类中比较重要的是法向量相关的信息，如RVertex和平滑组数据(smooth group)等。读者可以参阅参考资料。  
  
进一步的工作  
  
现在我们已经基本可以获得一个简单的骨骼动画系统所需要的大部分数据了。但是这仅仅局限于一个演示性的骨骼动画系统，离一个完整鲁棒的系统还有很多事情要做。  
  
上述例子中，我们仅仅导出了皮肤的顶点和第一层纹理坐标。读者还需要进一步处理多层贴图，顶点颜色，法向量等数据。在导出的骨架中，还应该能更方便的处理骨架的层次关系，以及能区分角色的上身和下身。因为通常一个骨骼动画系统是需要进行上下身的动作融合的。其次骨骼的变换矩阵应该是保存相对于父骨骼的局部变换矩阵，局部矩阵可以分解成四元数，平移和缩放，使用四元数能进行更平滑的插值和动作间的过渡，而且为了减少动作文件的大小，关键帧动画也是值得考虑的一个技术点。  
  
在数据组织方面，我认为一个完善的骨骼动画系统应该能合理而自然的管理所有数据，XReal3D的数据组织采用的是流式存储，所有的动作、骨架、顶点位置、纹理坐标都拥有自己的流。这样的存储方式，在保持弹性的前提下大大的减少了一个骨骼动画角色的文件数量，方便管理和维护。  
  
此外，现在的角色动画系统还应该包括表情动画和柔体系统。表情动画，一般称为在3DsMax的例子里有一个修改器就是表情动画，编译这个插件和使用它的头文件，我们可以访问Max里的表情动画。这个插件的位置在X:/3dsmax/maxsdk/samples/modifiers /morpher中。柔体系统也叫布料系统，可以用它来制作衣服飘动的效果。  
  
最后一个需要说的IGame， IGame是3DsMaxSDK中附带一个用来为导出3D游戏相关数据的开发包。基本上，IGame是3DsMAXSDK的一个包装，使用起来更加方便而已。要深入的了解和掌握3DsMax插件的开发，还是需要对3DsMaxSDK有一定的熟悉程度。  
  
总而言之，一个完整的骨骼动画系统插件是非常复杂的工程，鉴于文章篇幅，不可能介绍的面面俱到，希望本文对那些初次接触MAX插件开发的读者能有一定的帮助。   
  
参考资料  
  
1．3DsMAX Help  
  
2.  Cal3D :  https://gna.org/projects/cal3d/  
  
3.  XReal3D MAX Exporter : http://gforge.osdn.net.cn/projects/xreal3d/  
  
作者简介：  
  
潘李亮，2003年毕业于西北工业大学，爱好计算机图形学，曾在游戏公司负责3D引擎开发工作，目前在Corel公司从事多媒体软件开发。**

http://vipnews.csdn.net/newscontent.aspx?pointid=2008\_06\_03\_15114289