Ogre 数据文件结构分析

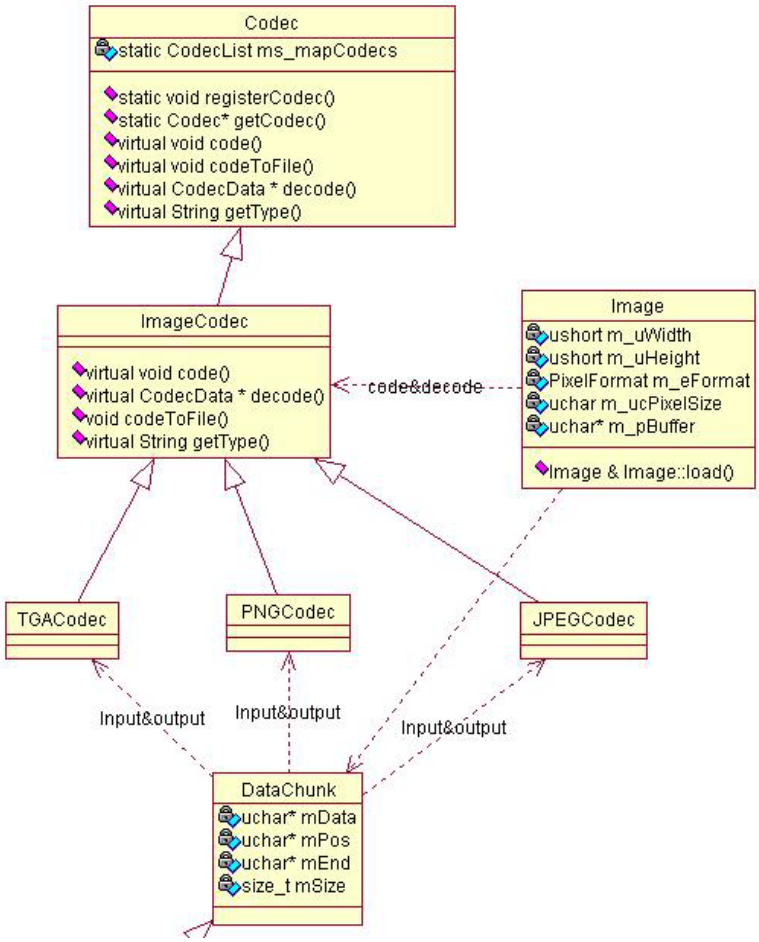
盛崇山
http://antsam.blogone.net
AntsamCGD@hotmail.com

Ogre 数据文件分析主要分析 Ogre 中用于存放外部文件相关的类，例如用于存放 Mesh 相关的类或者用于存放纹理的相关的类。这些类组成我们的道具^_^。我们将对一下几部分进行分析。

- 纹理
- CS 骨骼动画分析
- 模型（mesh）

一、纹理

纹理创建可以来之很多类型的外部文件，例如：bmp、jpg、png 等，而 DirectX（以 BMP 或 PNG 作为标准）和 OpenGL 的纹理正常情况是未经压缩的文件格式，所以 DirectX 和 OpenGL 中创建纹理所需要的数据也是没有压缩过的，所以对于 JPG、TGA、PNG 需要进行解压，我们可以利用第三方开发的库用来解压来获取数据。下面的 UML 图就是关于图像压缩相关的类：



图一：文件解压相关类

从图中我们可以看出纹理创建的过程大概如下流程：外部文件通过各类 Decode，把数据输出到 Data Chunk 中，然后 Image 通过 Load 函数把 Data Chunk 作为参数创建 Image 类，DirectX 和 OpenGL 再通过 Image 创建纹理。

二、模型

模型文件中比较重要是人物 Mesh，人物动画一般是通过关键帧或者骨骼动画实现的，现在的游戏大都采用骨骼动画。下面部分我们主要来分析骨骼动画。在分析 Ogre 中模型相关类前，我们先分析 Counter Strike 中的模型文件 MDL，MDL 是真正的骨骼动画。

在 MDL 文件中，定点数据只有一帧，作为初始位置，其它的帧通过骨骼动画来确定，其初始位置如下图所示：

图 2：模型初始位置点

初始位置是有了，那么，通过骨骼动画就可以动了，那么骨骼是怎么定义的呢？骨骼（Skeleton）当然是由很多的骨头（bone）组成的，我们看一下 Bone 的定义：

```
struct sBone
{
    char        name[32];
    struct      sBone *parent;
    float       matrix[3][4];
};
```

三个变量用来说明骨头的名称（大腿骨等^_^）、关节点、transform 矩阵；关节点把各个骨头连接在一起组成 skeleton（其实可以看成是一棵倒立的树型结构），还有一点必须说明的是：这种倒立的树型结构中父节点的动作会影响到子节点的，例如：大腿的运动会影响小腿的运动，也就是说 bone 中的 matrix 记录的是其相对于父节点的 transform，所以为了计算与 bone 相对应的定点的最后的位置，必须把这些定点对应的 bone 同其父节点、祖父节点等矩阵进行连接（contract）。

那么这些动作是怎么保存的呢？MDL 中动作称为 sequence,而每个动作(sequence)中又有许多关键帧（frame）组成，在骨骼动画中，关键帧是通过 skeleton 来表示，而在文件中则是记录 skeleton 中的每块 bone 的 transform，它们的定义如下：

```
struct sTransform
{
    float       pos[3];
    float       rot[3];
};

struct sSequence
```

```
    {
        char        name[32];

        float       fps;

        int         numframes;
        sTransform  *frames;
    };
```

sSequence中的transform的数目通过下面计算：

transform number = number of frame in sequence× number of bone in skeleton

而transform保存的则是bone相对与其父节点平移和旋转的信息。所以模型数据可以用下面的数据结构来表示：

```
  struct sModel
  {
    // 当前动作
    int         currentseq;
    // 当前所在的关键帧
    float       currentframe;

    // 在真个骨架中骨头的数目（^_^）
    int         numbones;
    // 指向bone的数据
    sBone       *bones;

    // 在模型中动作的个数
    int         numseqs;
    // 指向动作数据
    sSequence   *seqs;

    // 定点的个数
    int         numverts;
    // 指向定点数据
    sVertex     *verts;

    // 三角形个数
    int         numtriangles;
    sTriangle   *triangles;

    int         numtexture;
    sTexture    *textures[MAX_TEXTURES];
```

```
};
```

前面分析了关键帧是怎么保存的了，所以我们要渲染某个关键帧的话，我们必须设置好bone中的matrix，下面是在MDL loader中的代码：

```
void CMdlLoader::SetupBone()
{
    sBone   *pbone;
    float   q1[4],q2[4];
    float   m[3][4];
    sSequence   *pseq;
    sTransform  *ptrans;

    int     curframe;
    float   fraction;

    // 获取但前动作的首地址
    // get current frame and sequence;
    pseq = m_Model.seqs + m_Model.currentseq;

    // 计算关键帧之间的插值比例
    curframe = (int)m_Model.currentframe;
    fraction = m_Model.currentframe - curframe;

    // 获取骨骼的首地址
    pbone = m_Model.bones;

    for(int i = 0; i < m_Model.numbones; i++, pbone++)
    {
        // 获取但前frame，但前bone对应的transform的地址
        ptrans = pseq->frames + (pseq->numframes * i) + curframe;

        // 如果不是是最后一帧，着需要对两个关键帧之间进行插值
        if(pseq->numframes > (curframe + 1))
        {
            // 计算旋转、平移对应的矩阵
            AngleQuaternion(ptrans[0].rot, q1);
            AngleQuaternion(ptrans[1].rot, q2);
            QuaternionSlerp(q1,q2,fraction,q1);
            QuaternionMatrix(q1, m);

    m[0][3] = ptrans[0].pos[0] * (1.0 - fraction) + ptrans[1].pos[0] * fraction;
    m[1][3] = ptrans[0].pos[1] * (1.0 - fraction) + ptrans[1].pos[1] * fraction;
    m[2][3] = ptrans[0].pos[2] * (1.0 - fraction) + ptrans[1].pos[2] * fraction;
```

```
    }
    else
    {
        AngleQuaternion(ptrans->rot, q1);
        QuaternionMatrix(q1, m);
        m[0][3] = ptrans->pos[0];
        m[1][3] = ptrans->pos[1];
        m[2][3] = ptrans->pos[2];
    }

    // 如果当前的bone中是其他bone的子节点，则需要跟其父节点的矩阵
    // 进行连接，有一个疑问是如果其父节点又是别的bone的子节点
    // 那么如何保证其父节点的矩阵是已经连接好的矩阵呢？所以bone
    // 的保存顺序是有一定规律的，也就是说是从倒立树中的根节点，
    // 然后是其子节点，的顺序保存的（fixed me）
    if (pbone->parent)
        R_ConcatTransforms(pbone->parent->matrix, m, pbone->matrix);
    else
        memcpy(pbone->matrix, m, sizeof(float) *12);
}
}
```

好了，矩阵计算好了，那么我们需要把定点数据同矩阵相乘，都到transform后的定点位置，然后进行渲染。

Counter Strike的MDL骨骼动画分析的差不多了，现在我们可以看看Ogre中骨骼动画相关类的定义了。

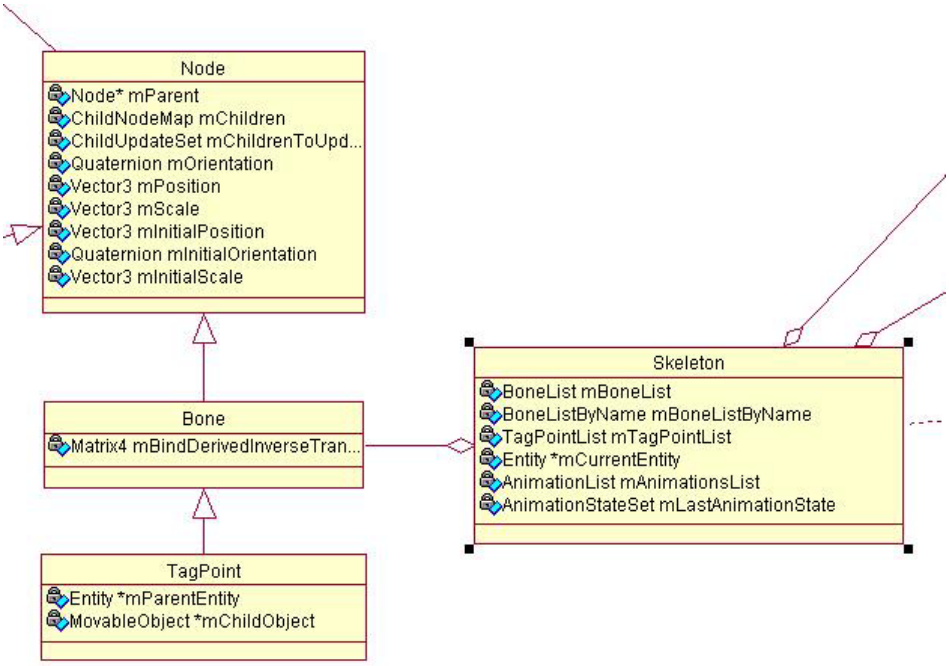
先看Ogre中骨骼（Skeleton）方面的相关类的UML图，图中的Node类在前面的文章《Ogre场景组织分析》中已经分析过了，其实就是一棵树型结构（好像不是像counter strike中的倒立树的结构）；现在我们看看Ogre中Bone的定义：

```
class _OgreExport Bone : public Node
{
      /// Pointer back to creator, for child creation
      /// (central memory allocation)
      Skeleton* mCreator;

      /// The inversed derived transform of the bone in the binding pose
      Matrix4 mBindDerivedInverseTransform;
```

这里的skeleton同counter strike中的区别是，counter strike中的skeleton是隐式的（通过bone的相互指针来表现），而Ogre中独立的定义了一个结构；而bone中还是保存了一个矩阵（同cs中式相同的^_^）；不过有一点不太明白的式为什么

保存的式Inverse 的matrix。



图三：Skeleton 相关类 UML 图

现在我们看看 Tag Point 的定义：

```
class TagPoint : public Bone
{
private:
    Entity *mParentEntity;
    MovableObject *mChildObject;
    mutable Matrix4 mFullLocalTransform;

 };
```

这个tag point的主要作用可能就是用于Attachment，例如：背包、枪支等。

下面看Skeleton的定义：

```
class _OgreExport Skeleton : public Resource
{
    SkeletonAnimationBlendMode mBlendState;
    /// Storage of bones, lookup by bone handle
    typedef std::map<unsigned short, Bone*> BoneList;
    BoneList mBoneList;
    /// Lookup by bone name
    typedef std::map<String, Bone*> BoneListByName;
    BoneListByName mBoneListByName;

    /// Storage of tagPoints, lookup by handle
```

```
    typedef std::map<unsigned short, TagPoint*> TagPointList;
    TagPointList mTagPointList;
    /// Entity that is currently updating this skeleton
    Entity *mCurrentEntity;

    /// Pointer to root bone (all others follow)
    mutable Bone *mRootBone;
    /// Bone automatic handles
    unsigned short mNextAutoHandle;

    /// TagPoint automatic handles
    unsigned short mNextTagPointAutoHandle;

    /// Storage of animations, lookup by name
    typedef std::map<String, Animation*> AnimationList;
    AnimationList mAnimationsList;

    /// Saved version of last animation
    AnimationStateSet mLastAnimationState;
```

上面的代码中蓝色代表的是同Skeleton相关类的成员变量，而红色代表的是同下面要分析的Animation相关的类的成员变量。蓝色部分我想已经比较清楚了，我们下面分析Animation部分。

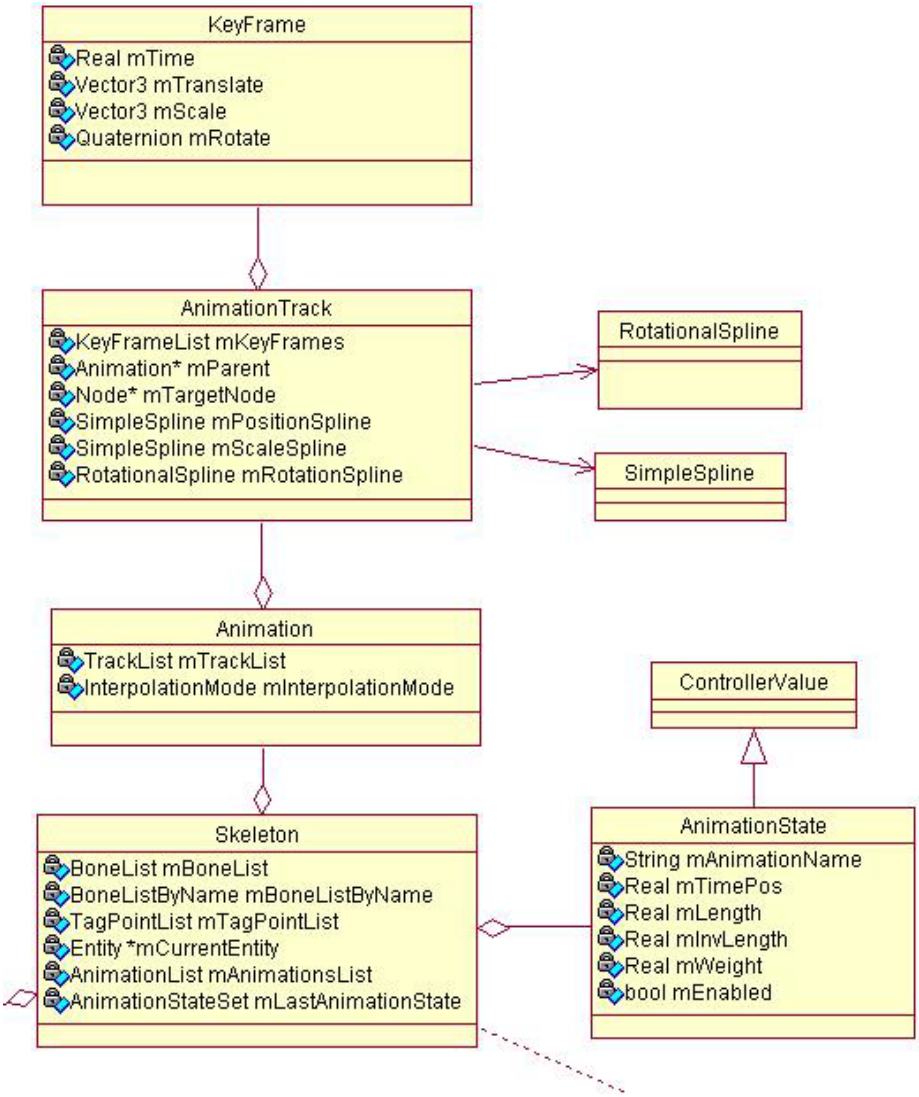
Animation（动画）相关类UML图如下所示，在骨骼动画中，其实可以把骨骼数据单独存放起来，这样就可以减少很多空间，只是外表不同，骨骼动画还是相同的。在图中的三个类Animation、Animation Track、Key Frame相当于MDL文件中的Model、Sequence和Frame。Skeleton的代码前面看过了，现在看Animation的定义：

```
class _OgreExport Animation
{
protected:
    /// Tracks, indexed by handle
    TrackList mTrackList;
    String mName;
    Real mLength;

    InterpolationMode mInterpolationMode;

    static InterpolationMode msDefaultInterpolationMode;
```

在Animation中保存了Track List（相当于Sequence动作），以及一些插值器，用当前Track的Key Frame之间进行插值。



Animation Track 定义如下：

```
/** A 'track' in an animation sequence, ie a sequence of keyframes which affect a
    certain type of animable object.
@remarks
    This class is intended as a base for more complete classes which will actually
    animate specific types of object, e.g. a bone in a skeleton to affect
    skeletal animation. An animation will likely include multiple tracks each of which
    can be made up of many KeyFrame instances. Note that the use of tracks
    allows each animable object to have it's own number of keyframes, i.e.
    you do not have to have the maximum number of keyframes for all
    animable objects just to cope with the most animated one.
@remarks
    Since the most common animable object is a Node, there are options in this
    class for associating the track with a Node which will receive keyframe updates
    automatically when the 'apply' method is called.
    class _OgreExport AnimationTrack
    {
```

```
public:
    typedef std::vector<KeyFrame*> KeyFrameList;
    KeyFrameList mKeyFrames;
    Real mMaxKeyFrameTime;
    Animation* mParent;
    Node* mTargetNode;

    // Flag indicating we need to rebuild the splines next time
    void buildInterpolationSplines(void) const;

    // Prebuilt splines, must be mutable since lazy-update in const method
    mutable bool mSplineBuildNeeded;
    mutable SimpleSpline mPositionSpline;
    mutable SimpleSpline mScaleSpline;
    mutable RotationalSpline mRotationSpline;
```

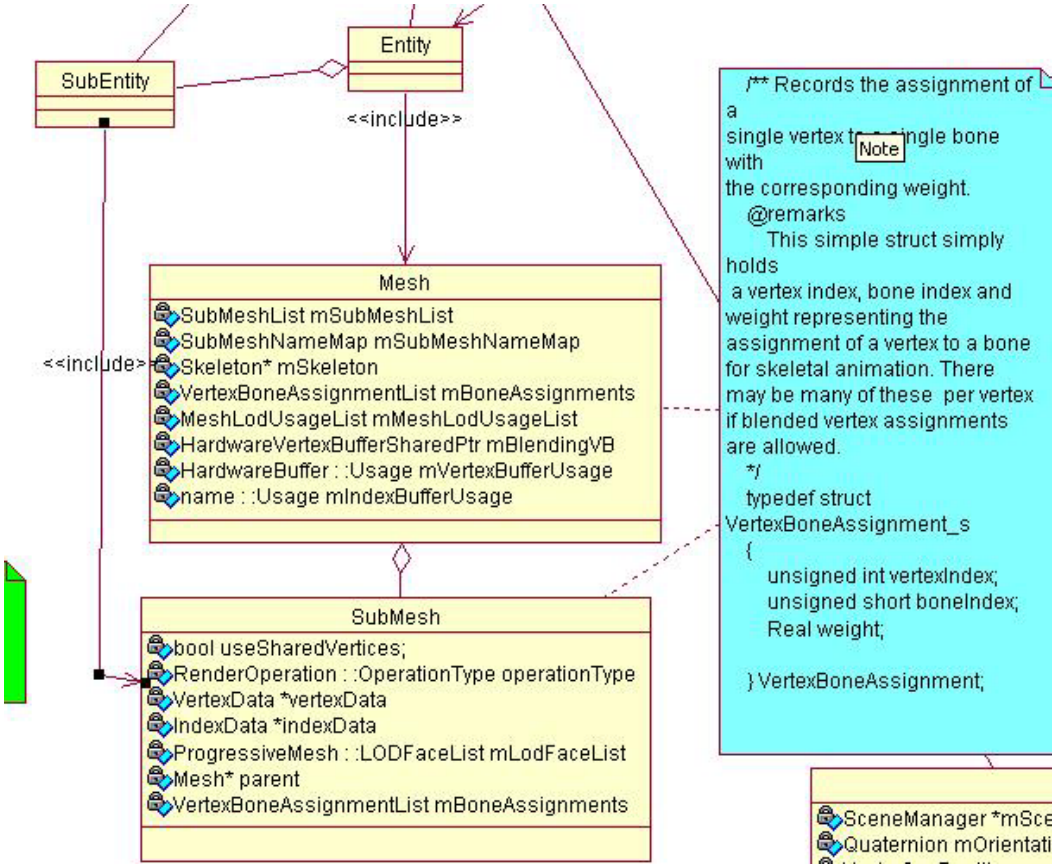
代码中用深红色表示的是关键帧数据，下面看一下 Key Frame 的定义，看看到底放了些什么东西。

```
class _OgreExport KeyFrame
{
public:
protected:
    Real mTime;
    Vector3 mTranslate;
    Vector3 mScale;
    Quaternion mRotate;
```

通过前面的 Counter Strike 骨骼动画的介绍，看到这些代码应该很清楚了吧，骨骼动画中，关键帧就是用 transform 来表示的：translate、scale、rotation。在 UML 图中，还有几个类 Spline 和 ControllValue，这些类主要是用于关键帧之间进行插值而引入的，这里不多介绍。

骨骼动画的原理以及 Ogre 中骨骼动画相关的类我们已经分析过了，现在看看 Ogre 中是怎么用这些类以及怎么实现骨骼动画的。

在 Ogre 中是通过 Mesh 来实现骨骼动画的，那么怎么把我们的 mesh 挂到场景树中去呢？现看看下面这张 UML 图：



从图中我们可以清楚的看出，Entity 和 Mesh，sub entity 和 sub mesh 的对应关系，现在我们可以挂到场景中了，那么 sub mesh 和 mesh 的关系是怎么样的呢？还是以 人物模型为例，整个人物模型，我们看成是一个 Mesh，那么模型中的各个部分：头、手，大腿、小腿、脚、躯干、脖子等（我们可以在分的细点）；那为什么要把模型给分出来呢？主要还是为了实现骨骼动画的需要，前面说过在骨骼动画中关键帧是通过记录 transform 来实现的，如果我们有 n 块 bone，那么每一个关键帧中有 n 个这样的 transform，如何组织可以参考前面的 Counter Strike 的分析。所以我们把整个 mesh 分成很多个 sub mesh，每个 sub mesh 对应一块 bone，那么关键帧就可以把 sub mesh 同相应的 bone 相乘来得到最终的位置，再进行渲染。

现在我们看看 Mesh 的代码：

```
class _OgreExport Mesh: public Resource
{
protected:
typedef std::vector<SubMesh*> SubMeshList;
/** A list of submeshes which make up this mesh.
    Each mesh is made up of 1 or more submeshes, which
    are each based on a single material and can have their
    own vertex data (they may not - they can share vertex data
    from the Mesh, depending on preference).
*/
SubMeshList mSubMeshList;
```

```
/** A hashmap used to store optional SubMesh names.
    Translates a name into SubMesh index
*/
typedef HashMap<String, ushort, _StringHash> SubMeshNameMap ;
SubMeshNameMap mSubMeshNameMap ;

/// Local bounding box volume
AxisAlignedBox mAABB;
/// Local bounding sphere radius (centered on object)
Real mBoundRadius;
/// Optional linked skeleton
String mSkeletonName;
Skeleton* mSkeleton;

VertexBoneAssignmentList mBoneAssignments;

/// Flag indicating that bone assignments need to be recompiled
bool mBoneAssignmentsOutOfDate;
bool mUseSoftwareBlending;

bool mIsLodManual;
ushort mNumLods;
typedef std::vector<MeshLodUsage> MeshLodUsageList;
MeshLodUsageList mMeshLodUsageList;

HardwareBuffer::Usage mVertexBufferUsage;
HardwareBuffer::Usage mIndexBufferUsage;
bool mVertexBufferShadowBuffer;
bool mIndexBufferShadowBuffer;
```

代码中的解释已经比较清楚了，代码中最后一部分主要是模型简化以及渲染所需的 buffer 的成员变量，现在还没有必要分析的那么仔细。还有一点是 Skeleton 中也是需要 setup bone 的^_^。现在看一下 Sub Mesh 的代码：

```
class _OgreExport SubMesh
{
    friend class Mesh;
    friend class MeshSerializerImpl;
    friend class MeshSerializerImpl_v1;

    /// Indicates if this submesh shares vertex data with other
    /// meshes or whether it has it's own vertices.
    bool useSharedVertices;
```

```
    /// The render operation type used to render this submesh
    RenderOperation::OperationType operationType;

    /** Dedicated vertex data (only valid if useSharedVertices = false).
        @remarks
            This data is completely owned by this submesh.
        @par
            The use of shared or non-shared buffers is determined when
            model data is converted to the OGRE .mesh format.
    */
    VertexData *vertexData;

    /// Face index data
    IndexData *indexData;

    ProgressiveMesh::LODFaceList mLodFaceList;

    /// Reference to parent Mesh.
    Mesh* parent;
    /// Name of the material this SubMesh uses.
    String mMaterialName;

    /// Is there a material yet?
    bool mMatInitialised;

    VertexBoneAssignmentList mBoneAssignments;
```

这部分代码中有三个方面需要说明的，第一部分就是开头的friends的类中有几个Serializer相关的类，serializer类的作用主要是读取文件，然后在把文件中的数据保存到Mesh实例中去 ;我想主要是因为保存在文件中的数据为了节省空间而玩了很多的trick，对数据进行压缩，所以不适合直接用mesh，需要进行必要的转换。第二部分则是数据部分，现在知道一下，还没有必要进一步分析下去的必要。第三部分最为重要，先看一席VertexBoneAssigmentList的定义：

```
typedef struct VertexBoneAssignment_s
{
    unsigned int vertexIndex;
    unsigned short boneIndex;
    Real weight;

} VertexBoneAssignment;
```

看到代码后是不是有一中霍然开朗的感觉^_^，对了，这样就把sub mesh同相应的bone联系在了一起，那么我们就可以都到最后的位置信息了（同bone相乘^_^）。

- Summary
  道具有了，我们可以排练节目了！^_^