3DEngine 技术文档

1. 资源管理类采用单例模式

单例模式的使用抄袭了 OGRE 的实现方式，即用一个单例模板类来实现。该类将拷贝构造函数和赋值函数设置成 private 成员，构造函数中采用 assert 来判断是否进行了创建，如果没有则正常执行构造函数，否则 assert 失败，程序会退出。

这样的设计的优点是，所有的资源管理类只有在需要的时候才会去创建，如果没有创建则会报错退出。同时，我们的资源管理子类就不需要考虑单例模式的问题，只需要提供一个构造函数和析构函数即可，同时对获取单例的函数进行重写。

我们创建资源管理类的策略是在 SceneManager 的构造函数中进行创建。

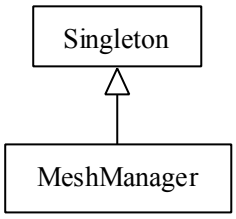

图 1 单例模式示意图

目前 MeshManager 是 Singleton 模式的一个例子，它负责所有 Mesh 的创建和销毁。为了能够复用 Mesh，我们创建了一个 map 来进行 Mesh 文件名到 MeshPtr 的映射，如果在 MeshManager 中已经存在了当前 Mesh 文件名的 MeshPtr，则直接返回；否则会创建这样的一个实例供用户使用。

对于上述提到的 MeshPtr，它是对 Mesh 进行封装的一个智能指针，采用 c++11（c++0x）的 std::shared_ptr 进行，所以编译时需要加上-std=c++0x 选项。采用对象对指针进行封装的好处是，我们只管 new，不管 delete。当所有的智能指针对象销毁后，其中的指针会被调用 delete 进行销毁。其中的基本原理是采用计数的方式。OGRE 的源码中使用了自己实现的智能指针。

2. SceneManager 以及渲染流程

我们的渲染必须在渲染窗口中才能进行，所以必须要和宿主操作系统建立管理建立一个可支持 OpenGL 渲染的窗口。这里我们目前采用的是第三方库的实现方案，采用的是 SDL（Simple DirectMedia Layer），这是一个跨平台的多媒体开发库，支持 Linux、Windows、Mac OS X 等，类似 Windows 环境中的 DirectX。它提供了图形、声音和用户输入输出的良好支持，目前这个项目仅仅使用了其图形和 IO 的功能。

未来，这个窗口的支持需要自己通过 EGL 来实现，因为我们未来的环境中没有图形系统的支持，我们的项目就是要构建一个图形系统的。关于 EGL 的配置是今后学习的一个东西。

创建摄像机是为了能够在场景中进行漫游，根据当前摄像头的位置渲染不同的场景。从这里开始就要接触关于数学方面的知识了。

首先是投影变换：

投影变换的目的是将视椎体变换成正规化空间（所有坐标范围从-1 到 1），OpenGL 的投影变换分成了两步，一步是设置投影变换矩阵进行变换，另一步是进行透视除法。为什么将透视除法单独提取出来的原因是，该变换和坐标的 z 值相关，而 z 值是始终在变化的。而我们希望的结果是它只和我们的摄像机的设置有关。所以，把透视除法作为投影变换后的一个步骤来进行。并且，该步骤是 OpenGL 状态机自动为我们执行的。

投影变换的四个参数：竖直方向的视角（FOV），宽高比（AR），进裁剪面（zNear），远裁剪面（zFar）。

最后的投影矩阵可以参考 sgi 的 gluPerspective 的实现，如下所示：

$$\begin{bmatrix} \frac{1}{\mathbf{AR}*\mathbf{tan}(FOV/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(FOV/2)} & 0 & 0 \\ 0 & 0 & \frac{zNear+zFar}{zNear-zFar} & \frac{2*zNear*zFar}{zNear-zFar} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

其次是视图变换：

视图变换的目的是将物体从世界坐标系中变换到摄像机的坐标系中，两个坐标系之间的转换直接使用线性代数中的基变换就可以解决，比较简单：

令 U 是摄像机空间中 X 轴在世界坐标系中的单位向量，V 是摄像机空间中 Y 轴在世界坐标系中的单位向量，N 是摄像机的 Z 轴在世界坐标系中的单位向量，则有：

$$\begin{bmatrix} Ux & Uy & Uz & 0 \\ Vx & Vy & Vz & 0 \\ Nx & Ny & Nz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} Xworld \\ Yworld \\ Zworld \\ 1 \end{bmatrix} = \begin{bmatrix} Xcamera \\ Ycamera \\ Zcamera \\ 1 \end{bmatrix}$$

所以需要用户给出 target 和 up 向量，我们在程序中默认的 target 和 up 是（0，0，-1）和（0，1，0），target 向量是摄像机空间中的-Z 方向，up 向量是摄像机中的 Y 方向。

这个也是参考 sgi 的源码，见 gluLookAt 函数实现。其中的 eye 表示当前 camera 的位置，center 表示朝向的位置，不要搞反了。

将这两个数学中的大难点搞定，其他的数学方面的计算就是比较小的了。摄像机负责渲染它所“看到”的场景：设置投影视图矩阵，根据渲染的节点设置模型矩阵，然后就是渲染节点上挂接的实体（Entity）。

基本渲染流程如下图：

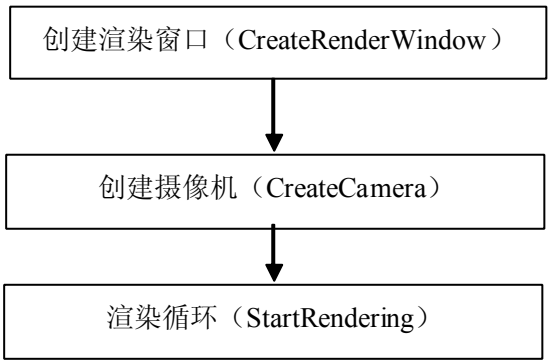

图 2 基本渲染流程

3. 场景节点模块（Node & SceneNode & SkeletonBone）

Node 类的功能较为单一但是不简单，它就是用来控制场景中的节点的位置信息的。记录一个节点的位置信息需要记录两组位置信息：一组是从世界坐标系变换信息，另一个是局部坐标系变换的信息。

世界坐标系变换信息会随着父亲节点的变动而变动。当本节点的位置信息发生移动时如果不通知其孩子结点，则其孩子结点无法知道父亲节点是否发生移动。这里的策略是:绘制循环中，显式调用_Update 函数，如果自身发生变动则主动通知孩子结点更改继承而来的位置信息。这个可以见 Camera 的 RenderNode 循环。

自身的变动始终是累加的，这种累加可以依据三种策略：TS_LOCAL. TS_PARENT, TS_WORLD。然后根据移动类型，又有不同策略，如下表所示：

	TS_LOCAL	TS_PARENT	TS_WORLD
移动	局部移动+=移动量*局部旋转	局部移动+=移动量	局部移动+=（移动量*父节点世界变换之旋转的逆方向）/父亲的全局缩放量
旋转	局部旋转=局部旋转*旋转量	局部旋转=旋转量*局部旋转	局部旋转=局部旋转*世界变换之旋转的逆旋转*旋转量*世界变换之旋转变换
缩放	局部缩放*=缩放量		

对其解释：

TS_WORLD 形式的 translate：在世界坐标系中进行移动，就应该按照原始的坐标轴进行移动，但是经过父亲节点的变换，全局坐标系已经进行了变换，如何恢复到原始的坐标系。只能乘以父节点变换的逆变换。Translate 只需要考虑方向，因此乘以的是父节点世界变换的逆旋转变换。

rotate 需要详细描述：

首先，旋转是用四元数来表示的，关于四元数的介绍可以参见相关文档。我的理解是用它表示旋转，可以利用乘法来表示两次连续的旋转，且插值也比较方便。如果仅仅记录三个角度的变换，无法描述上述的两次旋转合在一起的效果；矩阵的话，插值比较困难。

其次，两个四元数 a,b 相乘，a*b 表示 a 先起作用，b 后起作用。这个跟矩阵有用不同。所以，上面的 TS_LOCAL 是局部旋转*旋转量，表示局部变换先起作用，当前的旋转量后起作用；TS_PARENT 同样进行理解即可。

最后，TS_WORLD 形式需要好好理解：可以和 TS_LOCAL 进行对照，将“旋转量”变成“世界变换之旋转的逆旋转*旋转量*父节点世界变换之旋转变换”，前面的“世界变换之旋转的逆旋转”目的是变回原始的世界坐标系方向，然后再进行当前的旋转变动，最后在将“世界变换之旋转变换”的作用加回去。

每个节点的当前移动信息通过下表所述来生成，这个在渲染节点时用到，可以见 Node::updateFromParentImpl 函数实现。

移动量=（父节点世界缩放量*当前节点局部移动量*父节点世界旋转）+ 父节点世界移动量
旋转量=父节点世界旋转量*当前节点局部旋转量
缩放量=父节点世界缩放量*当前节点局部缩放量

其次是 SceneNode 类，这个类拿出来的目的是为了让他能够附加实体以便渲染，而骨骼节点则不需要附加实体。这个类对创建孩子结点的函数进行了重写。

SkeletonBone 继承 Bone 的目的是为了继承 Bone 的节点的父子关系，和每个节点之间的变换信息。SkeletonBone 与 Bone 的区别是，Bone 的父节点的变换会改变孩子结点的变换信息，但是 SkeletonBone 每个节点的变换信息是不变的，骨骼动画中存放的都是如同节点动画中的静态变换信息，只是在节点动画中以三个变换的形式存在，而这里以一个变换矩阵的形式存在。最后根据时间来进行插值，得到最后的变换信息。

场景节点模块的几个关键类的关系如下图所示：

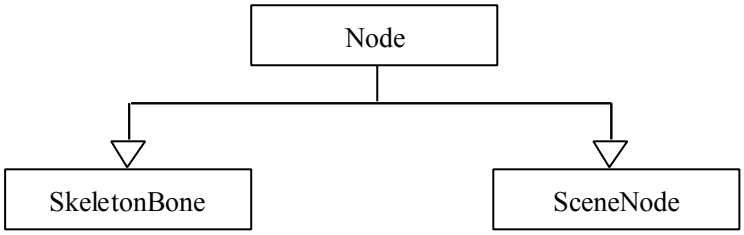

图 3 场景节点模块 UML 图

4. 动 画 模 块 （ AnimationState & Animation & AnimationTrack & KeyFrame）

动画的加入为场景的加入增色不少，目前我们的 3DEngine 仅仅支持两种类型的动画：场景节点动画和骨骼动画。骨骼动画的内容由于和实体的关系很大，因此和实体模块一起进行了介绍。这一小节仅仅介绍场景节点动画。

动画的控制分为两个部分，一个是时间控制的部分，这个是有 AnimationState 类来实现的；另一个部分是真正的动画部分，这个由另外的三个类来实现。

AnimationState 类的实现由于只管理时间相关的部分，因此它的实现较为简单：记录动画的总长度以及当前时间，动画是否循环播放以及是否使能。当给这个动画增加时间的时候，它就会通过计算得到动画当前的时间位置。

Animation 类是多个 AnimationTrack 类的集合，AnimationTrack 类是当前动画中关于某 SceneNode 的动画信息，包含了多个 KeyFrame，这些 KeyFrame 是该 SceneNode 的轨迹。

如何使得这些动画生效呢？有上面的 AnimationState 可以得到当前动画的时间位置，该时间位置用来给后来的 Animation 来使用，Animation 中的每个 AnimationTrack 利用该时间找到两个 KeyFrame，使得当前动画时间位置在这两个 KeyFrame 的时间值中间。再根据三个时间位置对两个关键帧进行插值，得到节点的偏移后对节点进行偏移即可。

动画模块的关键在于插值算法的实现，目前我的算法中采用的是最简单的线性算法，双线性插值以及弧线形插值目前还没有实现，需要进一步研究并加入到 3DEngine 中。

另外，需要保证一个节点当前只能参与一个节点动画，这个需要研究别人的代码如何实现。为何要作此限制呢？因为节点动画的实现是以动画开始之前的位置为基础的，如果同时执行两个动画，那么第一个动画做完后，第二个动画还是以第一个动画之前的位置为基础，从而忽视第一个动画的效果。最后的效果就是只有一个节点动画的效果会体现出来。

该模块几个类的 UML 图：

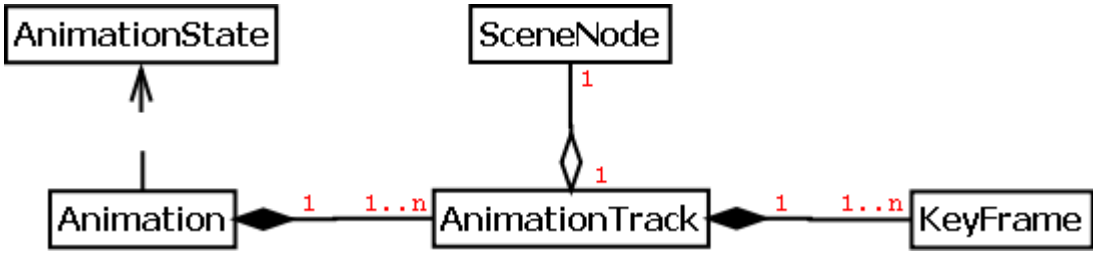

图 4 动画模块 UML 图

5. 实体模块（Mesh & SubMesh & Skeleton & Entity）

这个模块是和底层接触最多的模块，他们用来创建缓冲区对象并且利用缓冲区对象进行实际的渲染。下面对这几个类进行详细的介绍。

5.1 SubMesh

首先是 SubMesh 类，简单的理解，SubEntity 就是使用一张纹理贴图的一个区域。渲染 SubEntity 的过程就是设定一个纹理贴图，然后绘制 SubEntity 中的网格，这些网格是由三角形组成的。三角形则是由三个顶点组成的，每个顶点由多个字段进行描述，可看成一个多元组：坐标，纹理坐标，以及绑定的骨骼节点 ID 和权重 weight 数组，后来可能还会加入法线坐标和多种纹理坐标等。

SubMesh 是构成 Mesh 类的基础，整个的 Mesh 就是由多个 SubMesh 和与该 Mesh 相关联的 Skeleton 所构成的。

```
std::vector<Vector3f> coordVec; /*顶点数组，每个顶点是一个三维的数组*/
/*自己计算动画用的一个临时数组，计算好每个点的数据后用来改变顶点缓冲区*/
std::vector<Vector3f> finalCoordVec;
/*纹理坐标数组，纹理坐标是二维的，和上述顶点数组一一对应*/
std::vector<Vector2f> textureCoordVec;
/*每个顶点对应的骨骼的 id 和 wieight，最多对应四个。骨骼的数据有 Entity 进行保存，渲染时由它放入缓冲区，其中保存的是当前动画该骨骼节点的变换矩阵*/
std::vector<structAttachedBone> attachedBoneVec;
/*辅助数组，记录每个节点当前已关联的骨骼数量，以便加入*/
std::vector<unsigned> boneNumVec;
/*索引数组，这个和定点不是对应的了，渲染时主要用它*/
std::vector<int> indexVec;
//该 submesh 创建的三个 Buffer Object，有了他们就能进行绘制了
VERTEX_OBJ *vertexObject;
INDEX_OBJ *indexObject;
PIXEL_OBJ *pixelObject;
```

5.2 Skeleton 及骨骼动画

其次是 Skeleton 类，这个类只负责记录两个东西：骨骼节点的层次关系，以及由该

Skeleton 所形成的骨骼动画信息。成员如下所示：

```
/*记录了和该 Skeleton 相关的骨骼动画信息，动画信息是静态的*/
AnimationVector m_animationVec;
/*骨骼的层次结构*/
SkeletonBone *mp_rootBoneNode;
/*字符串到骨骼节点的 map，便于查找*/
BoneMap m_boneMap;
/*字符串到骨骼 id 的 map，骨骼的顺序在初始化 Mesh 时就确定了*/
BoneNameIndexMap m_boneNameIndexMap;
/*每个骨骼都有一个原始的的 offset 矩阵*/
BoneOffsetVector m_boneInfo;
/*全局的变换逆矩阵，作用未知*/
Matrix4f m_globalInverseMatrix;
```

上面两个类基本上就完成了骨骼动画的基础准备工作。骨骼动画看起来比较高级，其实内部的原理并不是很复杂，把它分成三个模块来想可能容易理解。

首先是骨骼信息的组织，这种组织结构类似于节点的组织，上面我们的实现也是采用公用父类的方式实现的，这种实现表示骨骼节点是一种层次关系的表示。骨骼节点与场景节点的一个不同点是，在组织关系上他们的层次关系的，但是在位置关系上他们并没有这层关系。就是说，parent 骨骼节点发生了移动，但是 child 骨骼节点则不受影响。为什么要这样实现呢？因为骨骼节点记录的移动并不是我们要的结果，我们也看不出来。我们要的是骨骼节点对我们的蒙皮顶点所产生的偏移，所以需要记住骨骼节点中的移动仅仅是自己的这种相对偏移，不需要考虑任何其他骨骼节点的移动对自己产生影响。

于是自然到了骨骼动画的第二个模块，骨骼节点与蒙皮顶点如何进行联系？这就跟 SubMesh 类中的一个元组有关：绑定的骨骼节点 ID 和权重 weight 数组。每个蒙皮顶点与几个骨骼节点通过一定的权重产生关联：每个骨骼节点的偏移矩阵*weight 对蒙皮节点进行变换。整个 Mesh 共享一份骨骼信息，这个信息保存在 Skeleton 类中。这需要对整个骨骼进行编号，并记录每个骨骼的偏移矩阵，这个矩阵数组记录在 Entity 类中。

最后一个模块就是骨骼动画了，所有的动画信息其实他们的整体框架都是类似的，这里的骨骼动画和上面介绍的场景节点动画其实就几乎是一样的。骨骼动画类由 SkeletonAnimation 类表示，每个 SkeletonAnimation 中有多个 SkeletonNodeTrack，这里的每个 track 是骨骼层次结构中一个骨骼节点在该动画中的 KeyFrame 的组合，SkeletonNodeTrack 与 NodeAnimationTrack 的区别是，SkeletonNodeTrack 中的所有 KeyFrame 中间的时间间隔是相同的，所以在 SkeletonNodeTrack 中不需要记录时间信息，计算当前时间到哪两个 SkeletonNodeTrack 之间时因此也不需要考虑 SkeletonNodeTrack，通过 SkeletonAnimation 就可以直接计算出来了。

5.3 Mesh

有了 SubMesh 类和 Skeleton 类作为基础，Mesh 的实现就变得相对简单。只是在初始化的时候需要完成对 SubMesh 类和 Skeleton 类的初始化文件读入过程。

其中的 SubMesh 初始化较为简单，将每个蒙皮顶点的坐标数组，纹理坐标数组，绑定的骨骼节点 ID 和权重 weight 数组，以及索引数组传递给 SubMesh 类即可。SubMesh 利用这些信息创建顶点缓冲，索引缓冲等。

初始化 Skeleton 分为两步，一步是初始化骨骼节点的层次关系，另一步是初始化骨骼动画。初始化骨骼节点的层析就是创建一个树形结构的骨骼节点。初始化骨骼动画则需要创建多个动画并保存在 Skeleton 中。

注意，初始化 Skeleton 的过程需要在初始化初始化 SubEntity 之前完成，因为读取 SubMesh 类的时候需要得到骨骼的 ID，而骨骼 ID 是存放在 Skeleton 中的。

```
std::vector<SubMesh*> m_subMeshes;/*Mesh 中的 SubMesh 数组*/
std::vector<Texture*> m_textures;/*纹理图像数组*/
unsigned m_numBones;/*骨骼数*/
Skeleton *mp_skeleton;/*骨骼信息*/
```

5.4 Entity

Entity 类是需要附着在 SceneNode 上的，为什么不是将 Mesh 附着在 SceneNode 上呢？因为，虽然 SceneNode 上附着的可能是相同的 Mesh，但是每个 Mesh 对应的动画进度可能不会相同。为了保证每个 SceneNode 上的动画相对独立，因此提取出了 Entity 类来封装 Mesh 和骨骼动画。

```
MeshPtr m_mesh;//对应的 Mesh
unsigned m_numBoneMatrices;//骨骼数量
BoneOffsetMatrixVector m_boneOffsetMatrixVec;//骨骼的变换矩阵数组
AnimationStateSet m_animationStateSet;//Entity 对应的骨骼动画集合
SkeletonAnimationMap m_animationMap;//map<name, SkeletonAnimaion*>
```

渲染 SceneNode 就是渲染附着在上面的 Entity，渲染 Entity 的过程如下：

1）在骨骼动画集合中遍历，找到当前使能动画，并将使能动画的偏移加到骨骼节点的偏移矩阵中（Entity::_updateAnimation（））；

2）将上述骨骼节点的偏移矩阵送到显存中；

3）渲染每个 Mesh；

下面附上一张图来本模块类的关系：（注意关联关系中的聚合和组合关系）

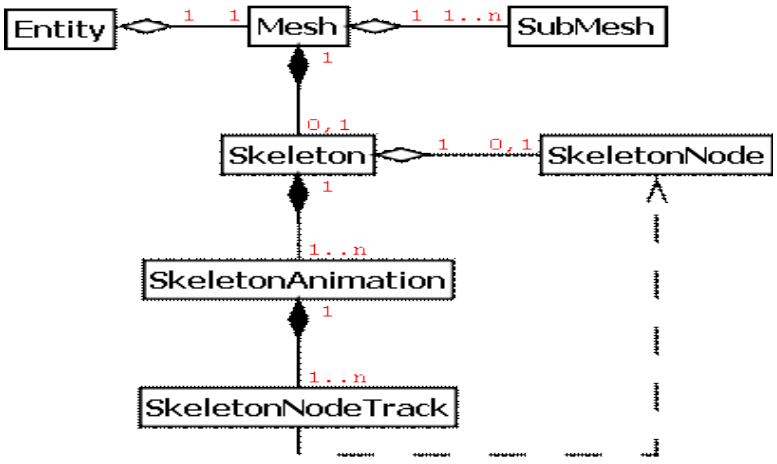

图 5 实体模块 UML 图