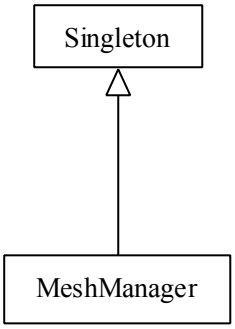
1. 资源管理类采用单例模式：

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。

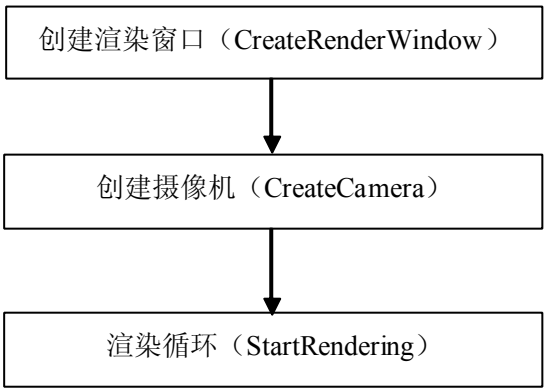
我们创建资源管理类的策略是在 SceneManager 的构造函数中进行创建。



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

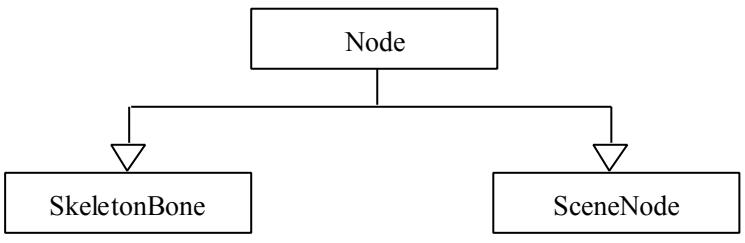
支持，我们的项目就是要构建一个图形系统的。关于 EGL 的配置是今后学习的一个东西。
创建摄像机是为了能够在场景中进行漫游，根据当前摄像头的位置渲染不同的场景。从这里开始就要接触关于数学方面的知识了。
首先是投影变换：
投影变换的目的是将视椎体变换成正规化空间（所有坐标范围从-1 到 1），OpenGL 的投影变换分成了两步，一步是设置投影变换矩阵进行变换，另一步是进行透视除法。为什么将透视除法单独提取出来的原因是，该变换和坐标的 z 值相关，而 z 值是始终在变化的。而我们希望的结果是它只和我们的摄像机的设置有关。所以，把透视除法作为投影变换后的一个步骤来进行。并且，该步骤是 OpenGL 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：

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

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



场景节点模块的几个关键类的关系如上图所示。

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
translate	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
rotate	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
scale	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。

4. 实体模块（Mesh & Entity）

5. 动画模块（Animation & AnimationState & AnimationTrack & KeyFrame）