OGRE 分析之场景管理

Mythma
http://www.cppblog.com/mythma
Email: mythma@163.com

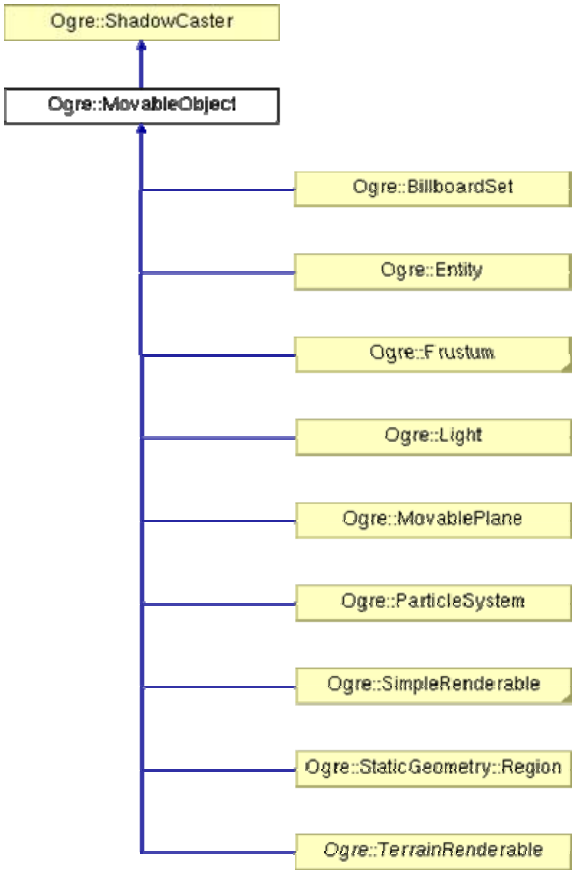
一、场景的组成

antsam在《Ogre场景组织分析》中比喻场景的组织：“场景组织就像一个舞台，需要摄影机、灯光、服饰、道具和演员”，在这里我也借用一下。场景大概有如下几部分组成：

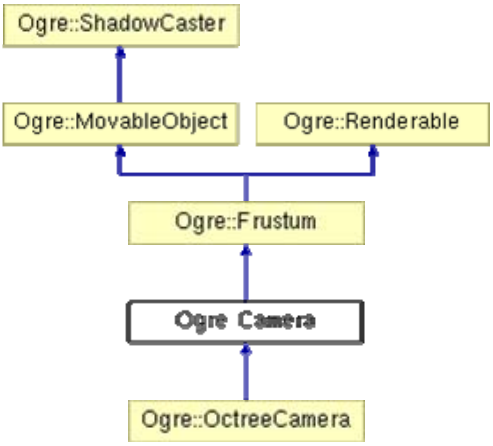
cameras ——〉摄影机
lights ——〉灯光
materials ——〉服装，不仅仅是演员的
entities ——〉演员
world geometry——〉房屋、地形等静态的场景
billboard
particle system
……

总体上来看，场景中的 objects 可分为可动实体和不可动实体。其中，Entities、Cameras 和 lights 属于 movable objects，World geometry 属于 immovable objects。materials 依附于别的 objects 之上，无所谓动与不动。

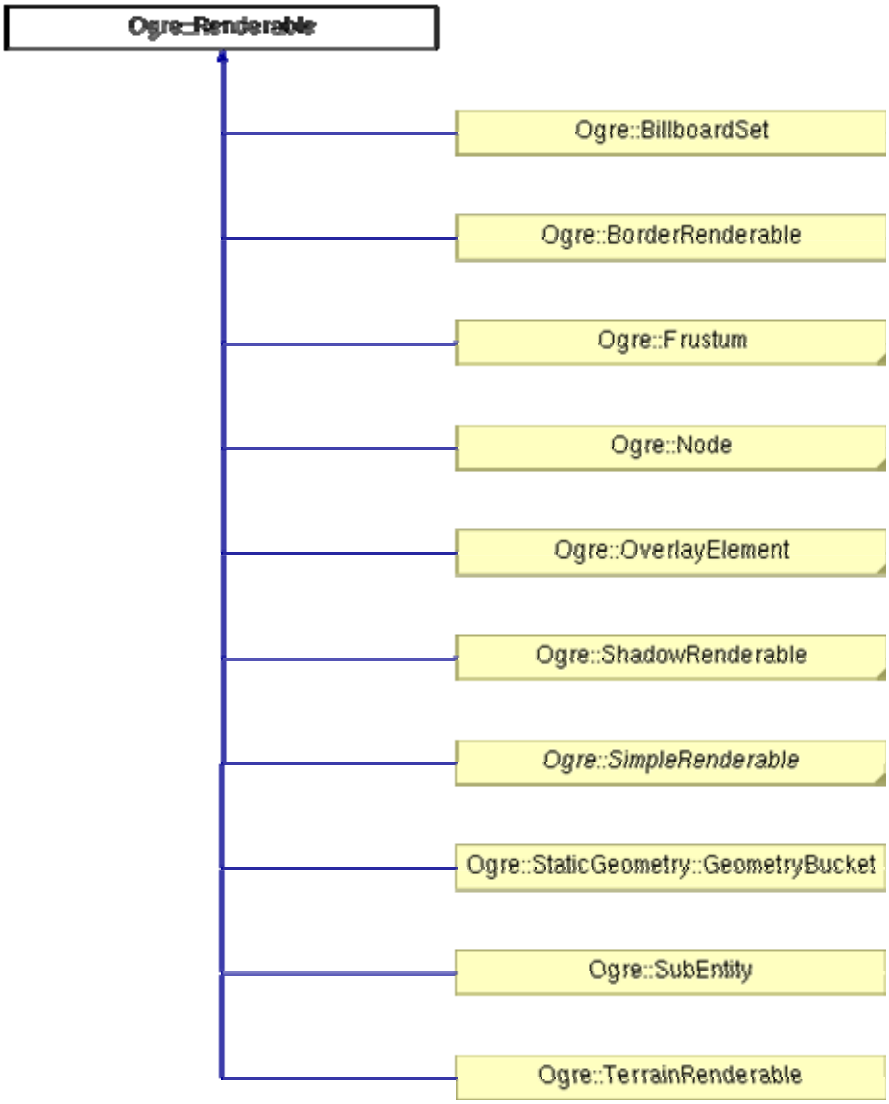
OGRE 把离散的，相对较小的，可移动的物体定义为 Ogre::MovableObject



Camera 属于 Frustum 的子类：

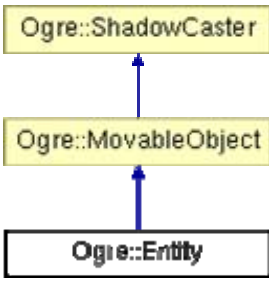


从另一个角度看，物体有可渲染和不可渲染之分。OGRE 中能被渲染的 Objects 都属于 Ogre::Renderable 类：



比较 MovableObject 和 Renderable 可以发现，我们的“主角”Entity 属于

MovableObject 但却不属于 Renderable：



Entity 是不可渲染的？那肯定不对。在 Renderable 有一个 SubEntity 的子类，而 Entity 是由 SubEntity 组成的：

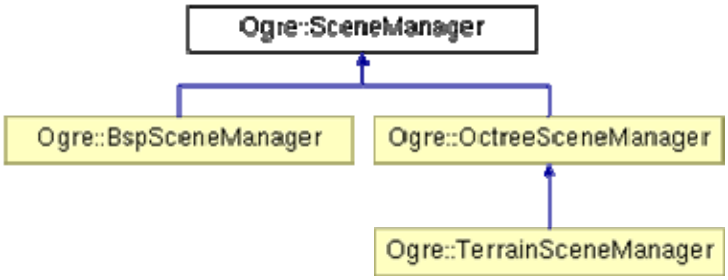
```
typedef std::vector<SubEntity*> SubEntityList;
SubEntityList mSubEntityList;
```

由此可以断定，Entity 是被分解为 SubEntity 完成渲染的。

二、场景的管理

1、Ogre::SceneManager

OGRE 的“场景管理员”是 Ogre::SceneManager。该类负责组织场景中的物体，并负责把物体发送到渲染系统进行渲染。SceneManager 本身是个虚基类，具体的实现由子类实现。



SceneManager 的数据成员较多，对于上述 Objects，基本上都用一个 List 来管理。另外，Camera, Light, SceneNode, Entity, BillboardSet, Animation, AnimationState, StaticGeometry 等创建都是通过 SceneManager 实现的，并且创建后直接放入相应的 List。

2、渲染队列

物体创建后，并不意味着就需要渲染在屏幕上。OGRE 把需要渲染的 Objects 放入一个渲染队列中——Ogre::RenderQueue。

1） RenderQueue 的组成

RenderQueue 由 Ogre::RenderQueueGroup 组成的（名字上好像有点别扭），RenderQueue 中有一个 RenderQueueGroup 的 Map 的数据成员：

```
typedef std::map< RenderQueueGroupID,
RenderQueueGroup * > RenderQueueGroupMap
typedef MapIterator< RenderQueueGroupMap > QueueGroupIterator
RenderQueueGroupMap mGroups
```

可见 RenderQueueGroupMap 的 key 为 RenderQueueGroupID，代表 Objects的渲染先后顺序。RenderQueueGroupID是一个枚举量，根据场景内物体的

渲染顺序由先及后定义，如 *RENDER_QUEUE_BACKGROUND* 为 0，*RENDER_QUEUE_OVERLAY* 为 100。

RenderQueue 通过成员 addRenderable 添加物体到渲染队列中：

```
void RenderQueue::addRenderable(Renderable* pRend,
                                RenderQueueGroupID groupID,
                                ushort priority)
{
    RenderQueueGroup* pGroup = getQueueGroup(groupID);
    // tell material it's been used
    pRend->getMaterial()->touch();
    pGroup->addRenderable(pRend, priority);
}
```

在 RenderQueue 的 getQueueGroup 成员负责 RenderQueueGroup 的查找创建。RenderQueueGroup 的生命周期由 RenderQueue 来控制。

2） RenderQueueGroup 的组成

RenderQueueGroup 中有一个 RenderPriorityGroup 的 Map 的数据成员：

```
typedef std::map<ushort, RenderPriorityGroup*, std::less<ushort> > PriorityMap;
typedef MapIterator<PriorityMap> PriorityMapIterator;
PriorityMap mPriorityGroups;
```

PriorityMap 的 key 为一个 ushort,它代表着 RenderPriorityGroup 渲染的优先级。对同一优先级的 Objects，RenderQueueGroup 会通过成员函数 addRenderable 将它加入相同的 RenderPriorityGroup 中:

```
void RenderQueueGroup::addRenderable(Renderable* pRend, ushort priority)
{
    // Check if priority group is there
    PriorityMap::iterator i = mPriorityGroups.find(priority);
    RenderPriorityGroup* pPriorityGrp;
    if (i == mPriorityGroups.end())
    {
        // Missing, create
        pPriorityGrp = new RenderPriorityGroup(this,
            mSplitPassesByLightingType, mSplitNoShadowPasses);
        mPriorityGroups.insert(PriorityMap::value_type(priority, pPriorityGrp));
    }
    else
    {
        pPriorityGrp = i->second;
    }
    // Add
    pPriorityGrp->addRenderable(pRend);
}
```

从上面代码可以看出，RenderPriorityGroup 的生命周期是由

RenderQueueGroup 管理的。

3） RenderPriorityGroup 的组成

RenderPriorityGroup 中是存放需要渲染的 Objects 的最终场所。需要渲染的 Objects——Renderable，RenderPriorityGroup 组织将其组织为 RenderableList，然后把 RenderableList 组织成 SolidRenderablePassMap：

```
typedef std::vector<Renderable*> RenderableList;
typedef std::map<Pass*, RenderableList*,
                 SolidQueueItemLess> SolidRenderablePassMap;
SolidRenderablePassMap mSolidPasses;
SolidRenderablePassMap mSolidPassesDiffuseSpecular;
SolidRenderablePassMap mSolidPassesDecal;
SolidRenderablePassMap mSolidPassesNoShadow;
```

综上所述，需渲染的物体分别经过 RenderPriorityGroup、RenderQueueGroup 分类后，由 RenderQueue 统一管理。

三、场景的组织

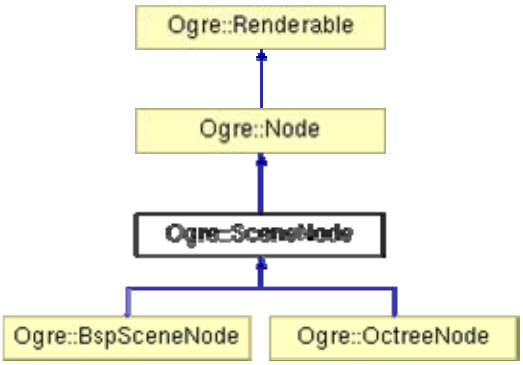
1、SceneNode

SceneManager 中有个重要的数据成员是 mSceneNodes：

```
typedef std::map<String, SceneNode*> SceneNodeList;
/** Central list of SceneNodes - for easy memory management.
 @note
  Note that this list is used only for memory management; the structure of the
  scene  is held using the hierarchy of SceneNodes starting with the root node.
  However yout  can look up nodes this way.
*/
SceneNodeList mSceneNodes;
/// Root scene node
SceneNode* mSceneRoot;
```

从注释中可以知道它的功能。

SceneNode 用来组织场景中的 objects，主要是管理 MovableObject。



从上面的图和 SceneManager 的类层次比较，可以发现很像**抽象工厂模式**的组织

结构。

SceneNode 的基类 Node 提供的如下方法就会建立一棵 Node 树：

virtual **Node** * **createChild** (const **String** &name, const **Vector3** &translate=**Vector3::ZERO**, const **Quaternion** &rotate=**Quaternion::IDENTITY**)
Creates a new named **Node** as a child of this node.

virtual void **addChild** (**Node** *child)
Adds a (precreated) child scene node to this node.

virtual unsigned short **numChildren** (void) const
Reports the number of child nodes under this one.

virtual **Node** * **getChild** (unsigned short index) const
Gets a pointer to a child node.

通过 SceneNodeList 就把各个棵 Node 树连接到一起，形成一片森林。

2、Entity 与 SceneNode：

有了森林，有了树木，有了树枝，但枝头是光秃秃的，毫无生机。而 Entity 就是点缀枝头的叶子、花朵、果实。

通过 SceneNode 的如下方法，OGRE 把 Entity 挂接到 SceneNode 上：

virtual void **attachObject** (**MovableObject** *obj)
Adds an instance of a scene object to this node.

virtual unsigned short **numAttachedObjects** (void) const
Reports the number of objects attached to this node.

virtual **MovableObject** * **getAttachedObject** (unsigned short index)
Retrieves a pointer to an attached object.

virtual **MovableObject** * **getAttachedObject** (const **String** &name)
Retrieves a pointer to an attached object.

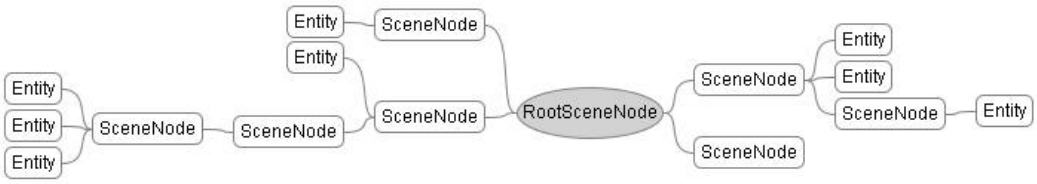
virtual **MovableObject** * **detachObject** (unsigned short index)
Detaches the indexed object from this scene node.

virtual void **detachObject** (**MovableObject** *obj)
Detaches an object by pointer.

virtual **MovableObject** * **detachObject** (const **String** &name)
Detaches the named object from this node and returns a pointer to it.

virtual void **detachAllObjects** (void)
Detaches all objects attached to this node.

关于Entity和SceneNode的关系，http://www.yanchen.com/ 画了这样一幅图，在此借用一下：



四、场景的建立

（略）