

# Ogre场景组织分析

盛崇山

<http://antsam.blogone.net>

[AntsamCGD@hotmail.com](mailto:AntsamCGD@hotmail.com)

场景组织是整个 Engine 的灵魂,而且到目前为止没有适用于任何场景的场景组织方式,所以都是以 Abstract Factory 方式进行组织,然后根据不同的场景采用不同的场景组织方式,再进行绘制。

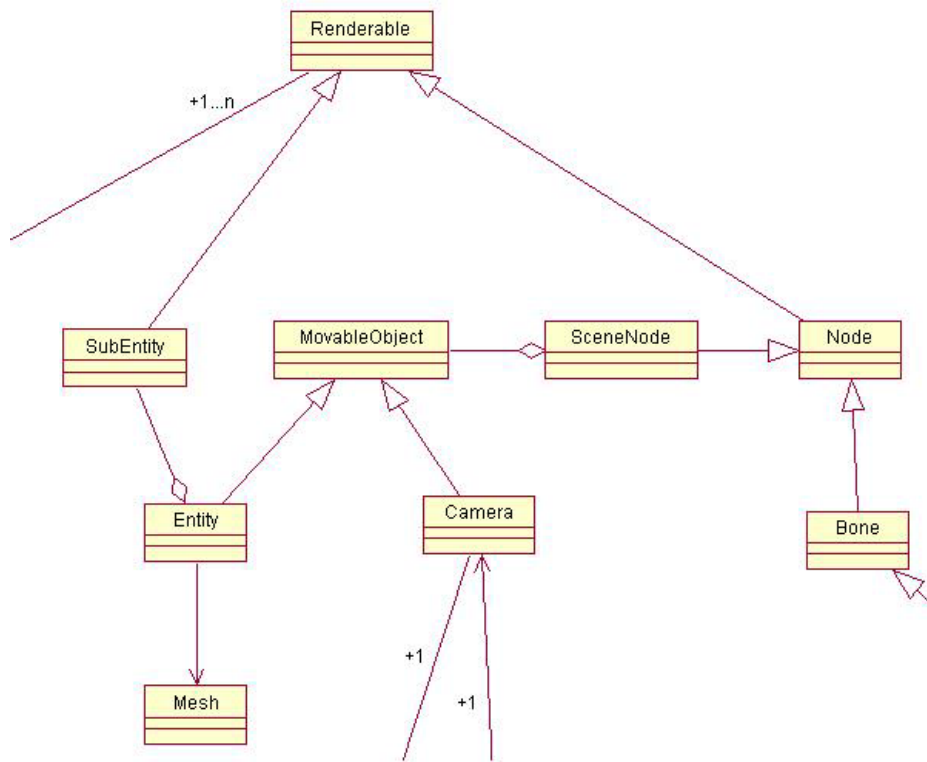
场景组织就像一个舞台,需要摄影机、灯光、服饰、道具和演员。那么摄影机就是 Camera,灯光就是 Light,服饰就是 Material,道具就是场景中的 Static Object,而演员就是 NPC。对于不同的场景需要不同的组织方式,才能有效的进行管理。但是对于不同的场景都有其基本特性。下面先从这些基本特性进行分析:

- 场景树
- 渲染队列
- camera 设置
- Light 设置

## 一、详细说明

- 场景树

以 Ogre 中的场景树为例进行分析,UML 图如下所示:



图一：Ogre 场景树有关的类 UML 图

在 UML 的顶部是一个抽象类 `Renderable`，作为场景中可以渲染 object 的父类，场景树中的每个节点都是可渲染的，所以都是从 `Renderable` 中集成而来。同场景树相关的类主要是 `Node`、`SceneNode`。下面我们可以看这两个类的部分代码进行分析。

Node 类：

```

class _OgreExport Node : public Renderable
{
public:
    typedef HashMap<String, Node*, _StringHash> ChildNodeMap;
    typedef MapIterator<ChildNodeMap> ChildNodeIterator;

protected:
    /// Pointer to parent node
    Node* mParent;
    /// Collection of pointers to direct children;
    ///hash map for efficiency
    ChildNodeMap mChildren;
}
  
```

从上面的代码中我们可以看出是一种属性结构。下面看 `SceneNode` 定义：

```

class _OgreExport SceneNode : public Node
{
public:
    typedef HashMap<String, MovableObject*, _StringHash> ObjectMap;
}
  
```

```
typedef MapIterator<ObjectMap> ObjectIterator;
```

protected:

```
ObjectMap mObjectsByName;  
/// Pointer to a Wire Bounding Box for this Node  
WireBoundingBox *mWireBoundingBox;
```

从代码中可以看出 SceneNode 类似于 Scene 中的一个 Cell，想象一下在 Octree Scene Manager 中的一个划分和 BSP 中的 cell。所以 SceneNode 可以认为是一个容器，可以存放 Movable Object。而存放在同一个 SceneNode 中的 object 应该有一定的共性，下面我们看一下 Node 中的其他一些代码：

```
/// Stores the orientation of the node relative to it's parent.  
Quaternion mOrientation;
```

```
/// Stores the position/translation of the node relative to its parent.  
Vector3 mPosition;
```

```
/// Stores the scaling factor applied to this node  
Vector3 mScale;
```

上面的代码可以知道存放在同一个 Scene Node (Cell) 中的物体具有相同的变换关系 (transform)。在介绍 Entity 后会举一个更具体的例子。

场景树已经定义好了，那么我们就可以在场景树上挂一下东西了 (^\_^有点像圣诞树啊)。下面我们看看用 Entity 来挂，Entity 的定义如下：

```
class _OgreExport Entity: public MovableObject  
{  
    // Allow SceneManager full access  
    friend class SceneManager;  
    friend class SubEntity;  
private:  
    /** The Mesh that this Entity is based on.  
    */  
    Mesh* mMesh;  
  
    /** List of SubEntities (point to SubMeshes).  
    */  
    typedef std::vector<SubEntity*> SubEntityList;  
    SubEntityList mSubEntityList;
```

也就说在 Scene Node 中可以存放各种 Mesh 和 sub entity。而 sub entity 定义为：

```
class _OgreExport SubEntity: public Renderable  
{  
    // Note no virtual functions for efficiency
```

```

friend class Entity;
friend class SceneManager;
private:
    /// Pointer to parent.
    Entity* mParentEntity;

    /// Name of Material in use by this SubEntity.
    String mMaterialName;

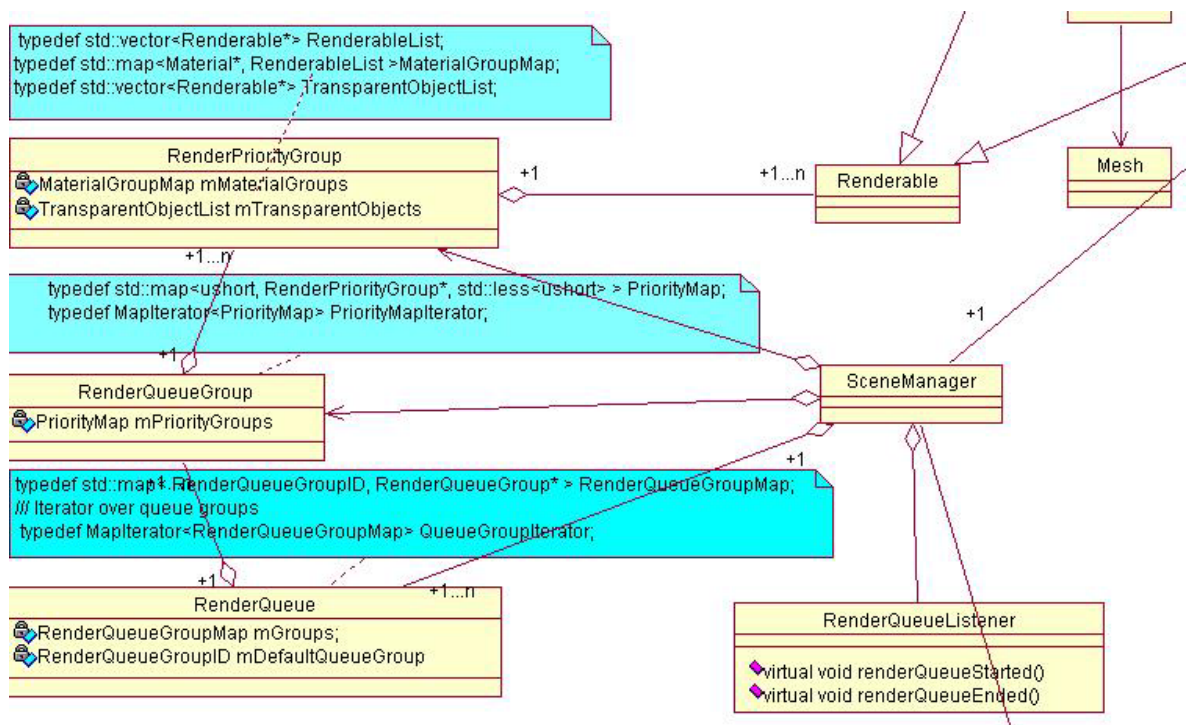
    /// Cached pointer to material.
    Material* mpMaterial;
    // Pointer to the SubMesh defining geometry.
    SubMesh* mSubMesh;

```

所以场景树上可以用来存放各种各样的 Mesh 或 sub Mesh。前面我们说过举一个更具体的关于同一个 Scene Node 中共性的例子。一个模型有很多个 Sub Mesh 组成，例如人物模型中的身体、躯干等，这些 sub mesh 必须是连接着的，也就是说其位置关系不变的，那么我们可以放在一个 Scene Node 中，因为它的 translation、rotation、scale 相对于父节点都是相同的。在 uml 图中我们还可以看到 bone 的定义，bone 是同 animation mesh 相关的，我们在《文件结构定义》中说明，还有就是 Camera 也可以挂靠到 Scene Node 中，这就可以把某些角色同 Camera 绑定，例如：第三人称游戏中 camera 的 control 方式。

- 渲染队列

前面我们分析了场景树，下面我们对渲染队列进行分析，其相关的类的 UML 图如下所示：



图二：渲染队列相关类 UML

从图中我们可以看出 render queue 是 Renderable 的集合，这是必然的，因为场景树和渲染队列其实都是对 Renderable 进行分类，只是分类的标准不同，场景树主要是从空间结构对 Renderable 进行分类，而渲染队列则是对 Renderable 从 material 以及 blend 上进行分类（渲染属性，从名字上可以推测出来）。

我们来看一下 RenderPriorityGroup 的定义：

```
class RenderPriorityGroup
{
    friend class Ogre::SceneManager;
    /** Comparator to order transparent objects
    */
public:
    typedef std::vector<Renderable*> RenderableList;
    /// Map on material within each queue group,
    /// this is for non-transparent objects only
    typedef std::map<Material*, RenderableList > MaterialGroupMap;
    /// Transparent object list, these are not grouped
    /// by material but will be sorted by descending Z
    typedef std::vector<Renderable*> TransparentObjectList;
protected:
    MaterialGroupMap mMaterialGroups;
    TransparentObjectList mTransparentObjects;
```

从成员变量的定义中可以看出 RenderPriorityGroup 对 Renderable 从 material 上进行了分类，具有相同的 Material 的 Renderable 分在同一个 list 中，渲染的时候可以一次性渲染用同一个 material 的 Entity，这样可以减少改变渲染状态的次数，以提高渲染速度（改变渲染状态 - pipeline config - 对速度影响非常大）。在成员变量中还有一个透明物体的 object list，这些物体是需要特别处理的。

在看一下 RenderQueueGroup 和 RenderQueue 的定义：

```
RenderQueueGroup :
class RenderQueueGroup
{
public:
    typedef std::map<ushort, RenderPriorityGroup*, std::less<ushort> >
        PriorityMap;
    typedef MapIterator<PriorityMap> PriorityMapIterator;
protected:
    // Map of RenderPriorityGroup objects
    PriorityMap mPriorityGroups;
```

```

RenderQueue :
class _OgreExport RenderQueue
{
public:
    typedef std::map< RenderQueueGroupID, RenderQueueGroup* >
        RenderQueueGroupMap;
    /// Iterator over queue groups
    typedef MapIterator<RenderQueueGroupMap> QueueGroupIterator;
protected:
    RenderQueueGroupMap mGroups;
    // The current default queue group
    RenderQueueGroupID mDefaultQueueGroup;

```

这两个类的定义主要是对不同的 group 进行分类，例如：当玩家在玩游戏的时候想推出，按了 esc 或者其他键，则会出现 2D 的 GUI 或其他 UI，这两种 Group（3D Group 和 2D Group）必须按一定的顺序渲染，否则结果将不可预测，也就是说它们的优先级不同，3D Group 比 UI 的优先级高，还有其他的 Group。上面的两个类就是这个用途。

上面已经分析了场景树和渲染队列，我们已经知道场景树和渲染队列都是对 `Renderable` 进行分类的，那么实例只存在一个，那么对其中的任何一个操作比较影响另外一个，两者必须交互，这个有待分析。

- Camera 设置等

Scene Manager 除了上面两个主要功能外，还有一些比较小的工作，例如 Camera 的设置，sky box 的设置（一般一个场景应该只有一个^^），fog 设置等。对于 fog 和 light 的设置要放到 Scene Manager 中而不放在 Render System 中，我想有一定的原因（废话^^），从 DirectX 的 API 中，一般渲染状态都是用 `SetRenderState` 等，而 Light 和 Fog 还有 Material 都有单独的函数：

- a) `SetLight`
- b) `SetMaterial`

不知道这样的解释是否合理，但是其实在 Render System 中实现也是可以的。

## 二、Scene Manager

上面的分析了一些 Scene Manager 的组成部分，那么 Scene Manager 的人物就是对这些组成部分进行操作了。先看一下 Scene Manager 的定义：

```

class _OgreExport SceneManager
{
protected:
    /// Queue of objects for rendering
    RenderQueue mRenderQueue;

```

```

    /// Current ambient light, cached for RenderSystem
    ColourValue mAmbientLight;

    /// The rendering system to send the scene to
    RenderSystem *mDestRenderSystem;

typedef std::map<std::string, Camera*, std::less<std::string> > CameraList;
typedef std::map<std::string, Light*, std::less<std::string> > LightList;
typedef std::map<std::string, Entity*, std::less<std::string> > EntityList;
typedef std::map<std::string, BillboardSet*, std::less<std::string> >
                                                BillboardSetList;

typedef std::map<String, SceneNode*> SceneNodeList;

    CameraList mCameras;
    LightList mLights;
    EntityList mEntities;
    BillboardSetList mBillboardSets;

    /** 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 you can look
            up nodes this way.
    */
    SceneNodeList mSceneNodes;

    /// Camera in progress
    Camera* mCameraInProgress;

    /// Root scene node
    SceneNode* mSceneRoot;

    // Sky params
    // Sky plane
    Entity* mSkyPlaneEntity;
    Entity* mSkyDomeEntity[5];
    Entity* mSkyBoxEntity[6];

    SceneNode* mSkyPlaneNode;
    SceneNode* mSkyDomeNode;
    SceneNode* mSkyBoxNode;

```

```

Quaternion mSkyDomeOrientation;
// Fog
FogMode mFogMode;
ColourValue mFogColour;
Real mFogStart;
Real mFogEnd;
Real mFogDensity;

```

上面用不同的颜色表示除了上面的各个组成部分。那么成员函数则是对这些组成部分的操作：创建、删除、设置以及获取当前属性等。当然这些只是每种 Scene Manager 都必须有的操作，而具体到不同的 Scene Manager（例如 BSP、Terrain 等）都有其自己特殊的 manager 函数。

- 创建函数

```

◆ createAABBQuery(const AxisAlignedBox &box, unsigned long mask = 0xFFFFFFFF)
◆ createAnimation(const String &name, Real length)
◆ createAnimationState(const String &animName)
◆ createBillboardSet(const String &name, unsigned int poolSize = 20)
◆ createCamera(const String &name)
◆ createEntity(const String &entityName, const String &meshName)
◆ createEntity(const String &entityName, enum SceneManager::PrefabType ptype)
◆ createIntersectionQuery(unsigned long mask = 0xFFFFFFFF)
◆ createLight(const String &name)
◆ createMaterial(const String &name)
◆ createOverlay(const String &name, ushort zorder = 100)
◆ createRayQuery(const Ray &ray, unsigned long mask = 0xFFFFFFFF)
◆ createSceneNode(const String &name)
◆ createSceneNode()
◆ createSkyboxPlane(enum SceneManager::BoxPlane bp, Real distance, const Qua
◆ createSkydomePlane(enum SceneManager::BoxPlane bp, Real curvature, Real tili
◆ createSphereQuery(const Sphere &sphere, unsigned long mask = 0xFFFFFFFF)

```

- 删除函数

- ◆ removeAllCameras()
- ◆ removeAllEntities()
- ◆ removeAllLights()
- ◆ removeBillboardSet(BillboardSet \*set)
- ◆ removeBillboardSet(const String &name)
- ◆ removeCamera(const String &name)
- ◆ removeCamera(Camera \*cam)
- ◆ removeEntity(const String &name)
- ◆ removeEntity(Entity \*ent)
- ◆ removeLight(const String &name)
- ◆ removeLight(Light \*light)
- ◆ removeRenderQueueListener(RenderQueueListener \*delListener)

- 设置函数

- ◆ setAmbientLight(ColourValue colour)
- ◆ setDisplaySceneNodes(bool display)
- ◆ setFog(FogMode mode = FOG\_NONE, ColourValue colour = ColourValue(0, 0, 0))
- ◆ setMaterial(Material \*mat, int numLayers)
- ◆ setOption(const String &strKey, const void \*pValue)
- ◆ setSkyBox(bool enable, const String &materialName, Real distance = 5000)
- ◆ setSkyDome(bool enable, const String &materialName, Real curvature = 10000)
- ◆ setSkyPlane(bool enable, const Plane &plane, const String &materialName)
- ◆ setViewport(Viewport \*vp)
- ◆ setWorldGeometry(const String &filename)

- 释放函数

- ◆ destroyAllAnimations()
- ◆ destroyAllAnimationStates()
- ◆ destroyAllOverlays()
- ◆ destroyAnimation(const String &name)
- ◆ destroyAnimationState(const String &name)
- ◆ destroyOverlay(const String &name)
- ◆ destroyQuery(SceneQuery \*query)
- ◆ destroySceneNode(const String &name)

- Get 函数

- ◆ getAmbientLight()
- ◆ getAnimation(const String &name)
- ◆ getAnimationState(const String &animName)
- ◆ getBillboardSet(const String &name)
- ◆ getBillboardSetIterator()
- ◆ getCamera(const String &name)
- ◆ getCameraIterator()
- ◆ getDefaultMaterialSettings()
- ◆ getEntity(const String &name)
- ◆ getEntityIterator()
- ◆ getFogColour()
- ◆ getFogDensity()
- ◆ getFogEnd()
- ◆ getFogMode()
- ◆ getFogStart()
- ◆ getLight(const String &name)
- ◆ getLightIterator()
- ◆ getMaterial(const String &name)
- ◆ getMaterial(int handle)
- ◆ getOption(const String &strKey, void \*pDestValue)
- ◆ getOptionKeys(std::list<String> &refKeys)
- ◆ getOptionValues(const String &strKey, std::list<SDDataChunk>)
- ◆ getOverlay(const String &name)

Scene Manger 的分析大致到此，还有一点必须注意的是类 :RenderQueueListener，其功能暂时还没有分析的必要。

### 三、Summary

场景组织 ( Scene Manager ) 好像更像导演，布置场景、灯光、服饰；同时还指挥着演员的表演方式；最后就是扛着摄影机进行拍摄（好像有专门的摄影的哦^\_^）。