OGRE （O－O Graphics Rendering Engine）

使用指南

V0.01a
最后更新于 **2003.4.10**

推荐使用 **Word 2000** 及以上版本阅读

大家看完之后如果有什么意见和建议请务必发 **Email** 提出，谢谢！！！

使用"查看"----"文档结构图"可大大方便阅读本文档

Mage 小组 著

Email: norman_chen@163.com
renwind@163.com
QQ: 18725262
http://www.173d8.com

OGRE（O－O Graphics Rendering Engine）

使用指南

作者

Mage 小组。Mage ＝ Make a game engine 或者 Marry a game engine ：-)
成员： Norman_chen norman_chen@263.net
noslopforever
renwind renwind@163.net
Windsprite ……

序

教程的目的

本教程的目的是从使用者的角度将 OGRE 引擎最基本的概念和使用方法做一个较全面

的介绍。本教程隐藏了 OGRE 引擎内部的底层内容，力求做到简单、易懂，是 OGRE 引擎的入门教程。

Mage 小组的另一篇教程＜＜OGRE 设计笔记＞＞正在整理中，它的目的是剖析 3D 图形引擎的源码，分析其架构，并在此过程中深入讲解设计模式与 OOA/D 在游戏开发中的应用。

读者要求

对游戏编程感兴趣。

虽然在使用 OGRE 过程中我们看不到被封装起来的图形 API（OGRE 支持所有的渲染 API）。但至少你使用用过一种图形 API（如 DirectX），并对图形编程有一定的认识。

虽然在使用 OGRE 过程中我们看不到被封装起来的操作系统 API（OGRE 支持所有的操作系统）。但是至少你要使用过系统 API（比如 Windows API）。

教程的由来

Mage 小组是由中国农业大学的几个在校学生组成的，我们由于共同的爱好起到了一起。

大二那年暑假（2002 年）我们三个人花了 40 天的时间写了三万行代码，做了一个游戏 DEMO（ http://cgd.pages.com.cn/cvbb/showthread.php?s=&threadid=942 ），通过这次实践，我们意识到了自己的差距：从来没有对游戏引擎做过设计，全部是硬编码（有 3 万行），这导致了游戏难于扩充与效率低下。

总结暑假的教训，我们意识到了游戏引擎的重要性。于是花了 4 个月的时间深入剖析了 OGRE（Object-Oriented Graphics Rendering Engine）的源码，做了大量实践。通过 OGRE 教程深入理解了设计模式，UML，面向对象分析与设计在游戏引擎设计中的作用。我们已经开始在 OGRE 引擎的基础上搭建自己的游戏引擎，充分体会到了面向对象图形引擎的灵活性与效率。

观察国内的游戏论坛，我们不难发现，有一些人认为学会图形 API 就可以编写出大型游戏，有些人在学过 DX 或 OGL 后不知该如何继续深入而放弃了游戏开发，论坛里对图形引擎的讨论少之又少。其实，图形引擎（OGRE 只是图形引擎）只占游戏引擎的 20%~30%，其它的还包括网络模块、物理（动力学）系统、人工智能（Game AI）、音乐等等。

现在我们把学习 OGRE 过程中的心得体会拿出来与大家分享，希望能引起大家对图形引擎，游戏引擎的注意。共同营造国内游戏程序员们的良好交流气氛。

OGRE 简介

OGRE（面向对象的图形渲染引擎）是用 C++开发的面向对象且使用灵活的 3D 引擎。它的目的是让开发者能更方便和直接地开发基于 3D 硬件设备的应用程序或游戏。引擎中的类库对更底层的系统库（如：Direct3D 和 OpenGL）的全部使用细节进行了抽象，并提供了基于现实世界对象的接口和其它类。

OGRE 特点

效率特性

- 简单、易用的面向对象接口设计使你能更容易地渲染 3D 场景，并使你的实现产品独立于渲染 API（如 Direct3D/OpenGL/Glide 等等）。
- 可扩展的程序框架（framework）使你能更快的编写出更好的程序。
- 为了节省你的宝贵时间，OGRE 会自动处理常见的需求，如渲染状态管理，hierarchical culling，半透物体排序等等。
- 清晰、整洁的设计加上全面的文档支持。

平台和 3D API 支持

- 支持 Direct3D 和 OpenGL
- 支持 Windows 平台，用 Visual C++ 6（或 Visual C++.Net）和 STLport 来编译。
- 支持 Linux 平台，用 gcc 3+（或 gcc 2.9x）和 STLport 来编译。
- 材质/Shader 支持
- 支持从 PNG、JPEG 或 TGA 这几种文件中加载纹理；自动产生 MipMap；自动调整纹理大小以满足硬件需求。
- 支持可程序控制的纹理坐标生成（如环境帖图）和转换（平移、扭曲、旋转）。
- 材质可以拥有足够多的纹理层，每层纹理支持各种渲染特效，支持动画纹理。
- 自动应用多通道渲染和多纹理，从而大幅度提高渲染质量。
- 支持透明物体和其它场景级别的渲染特效。
- 通过脚本语言可以不用重新编译就设置和更改高级的材质属性。

网格 Meshes

- 高效的网格数据格式
- 提供插件支持从 Milkshape3D 导出 OGRE 本身的.mesh 和.skeleton 文件格式。
- 支持骨骼动画（可渲染多个动画的组合）
- 支持用贝赛尔样条实现的曲面

场景特性

- 拥有高效率和高度可配置性的资源管理器，并且支持多种场景类型。使用系统默认的场景组织方法，或通过亲自编写插件使用自己的场景组织方法。
- 通过绑定体（如绑定盒）实现的场景体系视锥拣选。
- 提供的 BspSceneManager 插件是快速的室内渲染器，它支持加载 Quake3 关卡和 shader 脚本分析。
- 优秀的场景组织体系；场景结点支持物体的附属（attach），并带动附属物体一起运动，实现了类似于关节的运动继承体系。

特效

- 粒子系统包括可以通过编写插件来扩展的粒子发射器（emitter）和粒子特效影响器（affector）。通过脚本语言可以不用重新编译就设置和更改粒子属性。支持并自动管理粒子池，从而提升粒子系统的性能。
- 支持天空盒、天空面和天空圆顶，使用非常简单。
- 支持公告板，以实现特效。
- 自动管理透明物体（系统自动帮你设置渲染顺序和深度缓冲）

其它特性

- 资源管理和文档加载（ZIP、PK3）。
- 支持高效的插件体系结构，它允许你不重新编译就扩展引擎的功能。
- 运用'Controllers'你可以方便地改变一个数值。例如动态改变一个带防护罩的飞船的颜色值。
- 调试用的内存管理器负责检查内存溢出。

OGRE 中的模块

OGRE 中由很多模块组成，每个模块互相配合，共同实现 OGRE 的强大功能和优秀特性。OGRE 的模块大致可表现为如下结构，这也基本上是 OGRE 工程文件的结构：

OgreMain
PlatformManagers
SDL
Win32
Plugins
BspSceneManager
FileSystem
GuiElements
OctreeSceneManager
ParticleFX
RenderSystems
Direct3D7
Direct3D8
SDL
Tools
3ds2oof
3dsMaxExport
BitmapFontBuilderTool
MilkshapeExport

PythonInterface
XMLConverter

OgreMain 模块

特性	相关类
场景组织体系	Node, SceneNode, SceneManager, Camera, MovableObject
Material 管理	MaterialManager, Material, Material::TextureLayer
插件动态加载系统	Root, DynLibManager, DynLib
数学支持库	Math, Vector3, Matrix3, Matrix4, Quaternion
渲染器和几何管道	RenderSystem, RenderQueue, Renderable
网格/几何实体管理	MeshManager, Mesh, SubMesh, MeshSerializer, PatchSurface
资源管理	ResourceManager, Resource, ArchiveManager, ArchiveEx
天空/背景渲染	SceneManager
公告板系统和粒子系统	BillboardSet, Billboard, ParticleSystemManager, ParticleSystem, ParticleEmitter, ParticleAffector
日志和异常处理	Exception, LogManager, Log
事件监听器	FrameListener, RenderTargetListener
编解码器和图像加载器	Codec, JPGCodec, TGACodec, PNGCodec
自定义内存管理器	MemoryManager
基本动画	Animation, AnimationTrack, KeyFrame
骨骼动画	Skeleton, Bone, Animation, AnimationTrack, KeyFrame
字体渲染/字体加载	FontManager, Font
覆盖（Overlay）表面，二维元素	Overlaymanager, Overlay, GuiElement, GuiContainer

SDL 平台管理模块

这个模块通过 SDL 实现了非 Windows 平台的基本平台管理功能。

特性	相关类
输入管理	SDLInput
配置系统	SDLConfig

Win32 平台管理模块

实现了 Windows 平台的基本平台服务。

特性	相关类
输入管理	Win32Input
配置系统	Win32ConfigDialog

BSP 场景管理

该插件用 BSP 树和 clusters 提供了室内场景的管理。它可以导入 Quake3 的关卡。

特性	相关类
BSP 树	BspSceneManager, BspLevel, BspNode
关卡导入	Quake3Level
Shader 支持	Quake3Shadermanager, Quake3Shader

文件系统插件

提供在文件系统的文件夹中定位资源的能力。

GuiElement 插件

提供标准的二维表面元素，如文本输入区和边框。

OctreeSceneManager 插件

用八叉树管理标准场景。你也可以用它来渲染地形。

ParticleFX 插件

此插件提供了标准的粒子发射器和粒子特效影响器。

Direct3D7 渲染系统插件

此插件提供了基于 Direct3D7 的渲染系统。

Direct3D8 渲染系统插件

此插件提供了基于 Direct3D8 的渲染系统。

SDL 渲染系统插件

此插件提供了基于 OpenGL 和 SDL 的渲染系统。

3ds2oof 工具

这个工具可以将 3D Studio 的网络文件（.3ds）转换成.oof 格式（OGRE 以前的网络文件格式）。

这个工具已经被抛弃了。

3Dstudio Max 导出器

这是一个 3D Studio MAX(版本 4 或 5)的插件，可以将 3D Studio 的模型数据转换成 OGRE 的.mesh 或.skeleton 格式。

位图字体创建工具

能过此工具你可以把二进制字体文件转换成 OGRE 的.fontdef 文件。

Milkshape3D 导出器

这个工具是 Milkshape3D（一个建模工具）的插件，它允许你将模式导出成 OGRE 支持的.mesh 和.skeleton 文件格式。

Python 接口

设计这个子工程的目的是以 dll 的形式提供一个接口，使 Python（一种脚本语言）可以直接驱动 OGRE，并且允许 OGRE 直接调用 Python 脚本以实现游戏相关的脚本语言。

XML 转换器

此工具可以在.mesh/.skeleton 文件和 XML 文件之间互相转换。OGRE 引擎用二进制文件格式可以提升速度和效率，而这个工具允许你导出或导入 XML 文件，从而方便检查和修改（tweak）模型。

安装 OGRE

获取 OGRE

OGRE 是一个开放源码项目，该项目的网址是 ogre.sourceforge.net。在这里你可以获取到 OGRE 的最新版本和文档，此外你还可以在论坛上与其它开发者交流。

下载 OGRE 的最新版本，解压到硬盘上，你可以看到 OGRE 的工程文件夹和其它 DEMO 等内容。

支撑环境

OGRE 是一个比较大的项目，不可能每个功能都独立完成。OGRE 的编译和使用需要一些其它库作为支撑环境。在 Windows 环境下编译和安装 OGRE 需要如下支撑环境：

STLport 4.5.3

SGI 公司实现的 STL 跨平台版本。下载地址：http://www.stlport.org/ 。

安装方法：

解压缩 Download 下来的压缩文件

进入命令行环境，进入 STLport 的 src 目录下，依次执行以下命令。

copy vc6.mak makefile

nmake clean all

vcvars32.bat　注：这一步的作用是注册 vc 的环境变量

nmake install

以上命令执行完成后你会在 VC 的 include 文件夹中发现 STLport 子文件夹。同时 VC 的 lib 文件夹中也会出现如下文件：

stlport_vc6.lib

stlport_vc6_static.lib

stlport_vc6_stldebug.lib

stlport_vc6_stldebug_static.lib

Windows 系统文件夹会出现与以上文件相应的 dll。

为了防止 STLport 与 VC 自带的 STL 冲突，请在 VC 的 options 设置中将 include 路径加上安装后的 stlport 路径，且它应该放在搜索路径的最前面。

DirectX 8.1 SDK

微软公司的 DirectX 8.1 开发包。可从微软网站自由下载，标准安装界面。

安装完成后，在 VC 的 options 设置中将 include 路径加上安装后的 DirectX 8.1SDK 路径，它应该紧接在 stlport 路径之后。

其它第三方库

OGRE 引擎还用到了 jpeglib, libpng, zlib, and SDL 等几个库，分别提供各种图片文件、Zip 压缩文件访问能力和支持 SDL 平台。为了防止版本问题而出现错误，最好从 OGRE 网站上下载这些库，OGRE 将这些库打包成了名为 Dependencies 的压缩文件。下载完成后需要解包，并将其中的 ogrenews 文件夹下的 Dependencies 拷贝到 OGRE 的最新版本的 ogrenews 文件夹下。

编译 OGRE

在 VC6 环境里打开 OGREOGRE 的最新版本的 ogrenews 文件夹下的 Ogre.dsw 工作区文件，执行 Batch Build 指令，该指令会自动处理 OGRE 中各个工程的依赖关系，正确完成全部的编译构建。

运行 DEMO

在 ogrenew\Samples\Common\bin\Debug 下可以看到 Debug 方式编译的全部 DEMO。运行每一个 DEMO，感受 OGRE 应用程序的效果。当然这些都不是太完整的大场景，但对于学习 OGRE 来讲，每一个都是不错的练习，简单明了。

OGRE 运行期结构

运行完 DEMO 之后，注意查看 ogrenew\Samples\Common\bin\Debug 文件夹中的内容，从这里可以看到 OGRE 程序的运行环境。

除了 DEMO 的可执行文件外，该文件夹中还包括如下动态链接库：

OgreMain.dll	The core dll which includes all the basic classes, and abstractions of all the engine components which will be refined per OS platform, rendering API, or per scene type. This library must be on the path or in the current folder to be loaded at startup. The source for this is completely cross-platform.
OgrePlatform.dll	This library implements concrete versions of the classes required to be implemented for each platform. There are currently implementations for Win32 and Linux platforms. This library is loaded dynamically by the PlatformManager class (in OgreMain) and must either be on the path or in the current folder.
RenderSystem_Direct3D7.dll	Plugin library which adds a RenderSystem implementation for Direct3D. This library is loaded dynamically by the plugin architecture described above, and must be located in the folder

	specified in plugins.cfg. At the moment this uses Direct3D 7 interfaces, because the effort of migrating to v8 was not justified (the major addition, pixel shaders is still in it's infancy in terms of standards so hard to generalise), although more recent use of vertex blending in OGRE may change this stance. OGRE will definitely migrate to Direct3D 9 when released since this appears to be worth the effort, particularly the High-level shader language (HLSL).
RenderSystem_Direct3D8.dll	
RenderSystem_SDL.dll	Implementation of the RenderSystem which is based on both SDL and OpenGL. Used for the Linux platform primarily but can be used in Windows and because of the nature of SDL could be used on any platform OGRE is ported to.
Plugin_GuiElements.dll	
Plugin_BspSceneManager.dll	Plugin specialisation of the general-purpose SceneManager for rendering indoor levels based on a BSP tree. Allows Quake3 maps to be loaded and rendered very efficiently whilst exposing none of the complexity to the core Ogre system. This library is loaded dynamically by the plugin architecture described above, and must be located in the folder specified in plugins.cfg.
Plugin_OctreeSceneManager.dll	
Plugin_FileSystem.dll	Archive plugin that allows OGRE to read the filesystem of the target operating system.
Plugin_ParticleFX.dll	Plugin which provides a range of particle system tools such as standard emitters and affectors.
Devil.dll	
SDL.dll	
其它	你的机器上还必须包括 DirectX 8.1 和 STLport 的动态链接库，一般系统会将它们自动安装到 Windows 系统文件夹下，所以这里看不到，但它们也是必须的。

OGRE 的运行还需要如下的配置文件：

ogre.cfg	OGRE 的显示模式配置文件
Plugins.cfg	插件配置文件，在这里指定插件的路径和插件文件名。上一个表中以 Plugin_开头的 dll 文件都是插件，它们可以放在其它文件夹里，但必须在本文件里指定路径。
resources.cfg	资源配置文件，设置资源搜索路径，Zip 文件也作为搜索路径对待。
quake3settings.cfg	quake3 地图配置文件。
terrain.cfg	室外地形场景配置文件。

OGRE 程序的运行当然少不了各种资源，资源路径在 resources.cfg 里指定。OGRE DEMO 的资源都放在\ogrenew\Samples\Media 及其下的 Zip 文件里。资源文件包括以下内容：

.skeleton	骨骼动画的骨骼定义文件
.particle	粒子模板定义文件
.overlay	二维及三维界面定义文件
.mesh	模型文件
.material	材质定义文件
.fontdef	字体定义文件
.jpg	图片文件
.png	图片文件

准备用 OGRE 开发

从 FrameWork 开始

假设以 Demo_EnvMapping 工程为例，观察一下 OGRE 的 Demo 程序的代码结构，可以发现在本工程中其实只有两个文件：EnvMapping.h 和 EnvMapping.cpp。在 EnvMapping.h 中定义了应用程序类 EnvMapApplication，从字面上可以看出该类的 createScene（）函数负责创建场景，而后 EnvMapping.cpp 中创建该类的实例 app，并调用 app.go( )，渲染开始了！

真的这么简单吗？其实不然。app.go( )函数是在哪里定义的？仔细观察 EnvMapping.h 中对 EnvMapApplication 类的定义可以发现该类其实是一个派生类，它继承自 ExampleApplication 类，那么 ExampleApplication 类的定义在哪里呢？显然在 include 进来的 ExampleApplication.h 里。

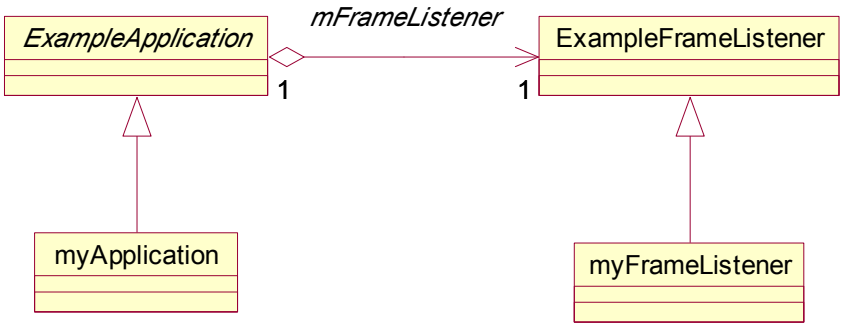
EnvMapping.h 头文件中，的 include 语句是这样写的：

#include "ExampleApplication.h"

可是，在 Demo_EnvMapping 工程的文件夹里并没有发现 ExampleApplication.h 文件存在，编译岂不是会出错！其实在 Demo_EnvMapping 工程中指定了附加包含路径。查看本工程的 Settings，在 C/C++选项卡中打开 PreProcessor 分类，可以看到有三个附加包含路径，其中..\include 是本工程的头文件路径，..\..\Common\include 是 Demo 程序公共的头文件路径，..\..\..\OgreMain\include 是 OGRE 引擎的头文件路径。在..\..\Common\include 中可以发现两个头文件：ExampleApplication.h 和 ExampleFrameListener.h。这两个文件定义了简单 OGRE 程序的应用框架（FrameWork），它们封装了简单 OGRE 程序的基本要素和运行过程。

FrameWork 与实际应用程序的关系

在创建我们自己的 OGRE 程序时，只需要继承 OGRE FrameWork 中的类并做少量改动就可以了。OGRE FrameWork 与实际应用的关系如下图所示：



OGRE FrameWork

ExampleApplication 类

OGRE FrameWork 的重要组成部分，该类实际上基本确定了一个 OGRE 应用程序的运行和开发方式。以下分析 ExampleApplication 类的构成，请结合实际代码一起分析。

数据成员

在 ExampleApplication.h 中定义了 ExampleApplication 类，该类定义了如下数据成员：

```
// 指向 Root 对象的指针
Root *mRoot;
// 指向程序中摄象机的指针
Camera* mCamera;
// 指向场景管理器的指针
SceneManager* mSceneMgr;
// 指向“帧监听器”的指针
FrameListener* mFrameListener;
// 指向渲染窗口的指针
RenderWindow* mWindow;
```

这些数据成员是一个 OGRE 应用程序必不可少的数据要素，以下依次进行解释：

Root：OGRE 系统的入口点

Root 对象在程序中必须最先创建和最后释放。OGRE 引擎是通过 Root 将其它部分“串”起来的，通过 Root 对象可以调出配置对话框以配置渲染系统（RenderSystem）；通过 Root 对象可以获取到引擎其它部分的指针，如 SceneManager、RenderSystem、Resource managers 等；Root 对象还提供一个 startRendering 方法来开始一个连续渲染过程，对该方法的调用在 ExampleApplication 类中就可以见到。

Camera：摄象机

渲染结果实际上就是摄象机最后“看”到的结果。

SceneManager：场景管理器

OGRE 将场景分为四种情况：普通场景、室外封闭场景、室外无限场景和室内场景，在引擎中定义了一个枚举型 SceneType 来代表场景类型，该枚举型有四个枚举元素分别对应四种场景：

```
enum SceneType
    {
        ST_GENERIC,
        ST_EXTERIOR_CLOSE,
        ST_EXTERIOR_FAR,
        ST_INTERIOR
    };
```

OGRE 引擎针对不同的场景采用不同渲染策略，如针对室内场景可采用 BSP 场景管理方式，针对室外场景可采用八叉树场景管理方式等。我们可以采用编写插件的方式来扩展和指定不同场景的渲染策略。当然在刚开始 OGRE 编程的时候，使用 ST_GENERIC 普通场景类型就可以了。

需要注意的是：对于复杂室内和室外场景，OGRE 将场景分为两大部分，一部分是基本上固定不变的“世界”，对于这部分采用 BSP 或 Octree 等特殊的技术和算法来提高渲染效率。另一部分是场景中的可移动物体，这部分物体的创建和控制还需要开发人员自己来完成。

FrameListener：帧监听器

监听最终用户的控制信息（鼠标、键盘、遥控杆等），对摄象机、场景物体等进行控制，详细情况见后面对 FrameListener 的讲解。

RenderWindow：渲染窗口

渲染结果所在的窗口，其中包括渲染真正的目的地：视口 Viewport。

成员函数

在 ExampleApplication 类中定义了如下公共成员函数：

构造函数（略）

析构函数（略）

go 函数

virtual void go(void)：

该函数是 ExampleApplication 类除构造和析构函数以外唯一的 public 函数，它一调用，程序就正式开始运行了。代码如下：

```
virtual void go(void)
{
    if (!setup())
        return;

    mRoot->startRendering();
}
```

从代码可以看出，先调用 setup( )函数完成渲染前的准备，如果 setup 成功，就由 mRoot 调用 startRendering()开始渲染，如果不成功则退出。

setup 函数

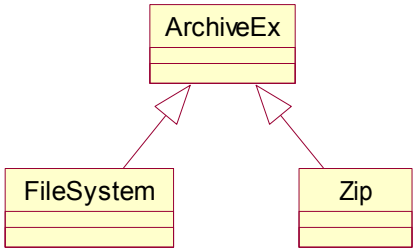
Setup 函数完成一个 OGRE 应用程序的开始渲染前的准备工作。代码如下：

```
virtual bool setup(void)
    {
      mRoot = new Root();
      setupResources();
      bool carryOn = configure();
      if (!carryOn) return false;
            chooseSceneManager();
            createCamera();
            createViewports();
            // Set default mipmap level (NB some APIs ignore this)
            TextureManager::getSingleton().setDefaultNumMipMaps(5);
            // Create the scene
            createScene();
            createFrameListener();
            return true;
      }
```

实际步骤如下：

- 首先创建 Root 类的对象。
  Root 对象必须在 OGRE 其它对象创建之前创建
- 加载资源路径 setupResources(void)
  资源是 OGRE 应用程序渲染过程中需要用到的纹理图片、网格模型文件、骨骼动画文件的总称。OGRE 应用程序需要在渲染前将这些资源载入内存，那就需要让 OGRE 引擎知道资源的搜索路径。特别的是 OGRE 引擎支持直接读取 Zip 压缩文件中的内容，所以 Zip 文件也必须被当成搜索路径来指定。在 OGRE 引擎中具有虚拟文件系统的概念，引擎内部载入资源文件都是通过虚拟文件系统来进行的，引

擎并不关心资源文件来自一个普通文件夹、zip 压缩包甚至网络映射。真正的文件读取功能是通过插件来实现的，所以大家在运行环境里可以发现 Plugin_FileSystem.dll，早期的 OGRE 版本（0.99b）还有 Plugin_Zip.dll，在新的版本里被实现到引擎内部了。目前还没有实现对网络文件的直接访问。



虚拟文件系统

为了方便 OGRE 程序在运行期间查找资源，使用了资源配置文件 resources.cfg。这是一个文本文件，我们可以在 OGRE 程序的可执行文件的同一文件夹下找到它。它的内容就是对资源路径的指定，示例如下：

```
Zip=../../../Media/dragon.zip
Zip=../../../Media/knot.zip
Zip=../../../Media/skybox.zip
FileSystem=../../../Media/
```

如果资源在一个 Zip 文件中，就写 Zip=******，如果资源在一个普通的硬盘文件夹里就写 FileSystem=******，通常这两种情况都有。例如在 OGRE 自带的 Demo 中，就将大部分资源放在一个文件夹里，特殊的资源该文件夹中的 Zip 文件里。

setupResources 函数就在 ExampleApplication 里，是一个 protected 函数。代码如下：

```
virtual void setupResources(void)
    {
        // Load resource paths from config file
        ConfigFile cf;
        cf.load("resources.cfg");

        // Go through all settings in the file
        ConfigFile::SettingsIterator i = cf.getSettingsIterator();

        String typeName, archName;
        while (i.hasMoreElements())
        {
            typeName = i.peekNextKey();
            archName = i.getNext();
            ResourceManager::addCommonArchiveEx( archName, typeName );
        }
    }
```

它首先创建并载入资源配置文件 resources.cfg，而后定义一个迭代器，通过循环的方式将配置文件中的内容（实际上是不同类型的资源搜索路径）全部读出，并加入到 ResourceManager 的档案搜索路径中。ResourceManager 是 OGRE 引擎中的资源管理器类，它是其它纹理、模型等各种资源管理器的基类，

addCommonArchiveEx 函数是该类的静态成员函数，用于加入各种资源的公共搜索路径，当然各种资源管理器也可以有自己的特殊搜索路径，在此不再详述。有关 ConfigFile 类的使用方法见后详述。

- 弹出 config 对话框，配置 RenderSystem

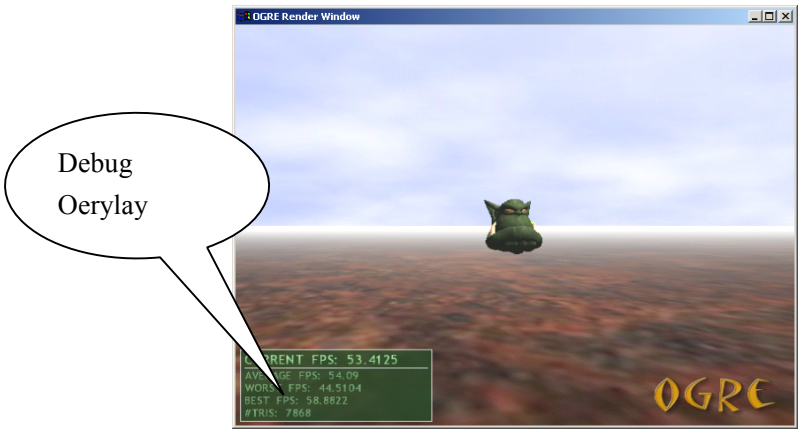
setup 函数通过如下代码弹出对话框，并判断如果用户没有点击对话框中的确定按钮则程序退出。

```
bool carryOn = configure();
if (!carryOn) return false;
```

configure()函数是 ExampleApplication 类中的 protected 类型函数，代码如下：

```
virtual bool configure(void)
    {
        // Show the configuration dialog and initialise the system
        // You can skip this and use root.restoreConfig() to load configuration
        // settings if you were sure there are valid ones saved in ogre.cfg
        if(mRoot->showConfigDialog())
        {
            // If returned true, user clicked OK so initialise
            // Here we choose to let the system create a default
            // rendering window by passing 'true'
            mWindow = mRoot->initialise(true);
            mRoot->showDebugOverlay(true);
            return true;
        }
        else
        {
            return false;
        }
    }
```

程序通过 Root 对象的指针调用 showConfigDialog()函数弹出对话框，在该对话框中最终用户可以选择使用何种渲染系统（OGRE 支持 DirectX 的 7.0 以上版本和 OpenGL）、是否使用全屏渲染以及渲染分辨率等。如果最终用户选择 OK 按钮，程序调用 Root 对象的 initialise 函数，在其内部初始化渲染系统、创建并初始化窗口而后返回窗口指针并赋值给 ExampleApplication 类的 mWindow 成员。紧接着程序调用 Root 对象的 showDebugOverlay 函数显示渲染窗口的 DEBUG 信息框（也就是大家所见的显示 FPS 情况的信息框）。



DebugOverlay

需要注意的是：OGRE 引擎将会在可执行文件所在的文件夹中生成一个 ogre.cfg 文件，该文件保存了本次对话框的设置结果，下次显示对话框时会将上次的配置结果读入并直接显示。如果你认为在程序运行之前显示一个对话框很难看，或者希望采用另外的形式（比如专门的配置界面，就象在大部分游戏中那样）来配置渲染系统，只要你确保在可执行文件的同一文件夹下存在有效的 Ogre.cfg 配置文件，就可以直接调用 Root 对象的 restoreConfig()函数来直接读入渲染系统配置信息，而不显示配置对话框。

- 选择场景管理器类型

```
mSceneMgr = mRoot->getSceneManager(ST_GENERIC);
```

通过调用 Root 对象的 getSceneManager 函数选择场景类型并赋值给 ExampleApplication 类的 mSceneMgr 成员，这里仅仅选择了普通场景类型。对于复杂的场景可以使用 ST_EXTERIOR_CLOSE、ST_EXTERIOR_FAR、ST_INTERIOR 等参数来选择室外或室内场景管理器，在 OGRE 引擎中提供了 Plugin_BSPSceneManager.dll 和 Plugin_OctreeSceneManager.dll 两个插件，分别采用 BSP 树和 Octree（八叉树）技术来对室内、室外固定场景实现高效渲染。场景管理器一旦确定也就确定了场景的组织管理和渲染方式。

- 创建并初始化摄像机

createCamera 函数是 ExampleApplication 类的 protected 类型成员。代码如下：

```
virtual void createCamera(void)
    {
        // Create the camera
        mCamera = mSceneMgr->createCamera("PlayerCam");

        // Position it at 500 in Z direction
        mCamera->setPosition(Vector3(0,0,500));
        // Look back along -Z
        mCamera->lookAt(Vector3(0,0,-300));
        mCamera->setNearClipDistance(5);

}
```

Root 对象负责调动引擎的各个子系统来完成渲染，但场景管理器负责管理和组织场景，摄象机属于场景中的可移动物体，所以摄象机的创建是由 mSceneMgr 成员来

完成的。mSceneMgr 调用 createCamera 函数来完成摄象机的创建，该函数的参数是摄象机的名字。由于摄象机需要接收控制信息，所以将已创建的摄象机的指针赋值给 ExampleApplication 类的 mCamera 成员，以便于对其进行控制。

摄象机创建完成后，初始化其位置、观察方向和平截台体的近面距离。

- 创建窗口中的视口

窗口的创建和初始化是在显示完配置对话框之后由 Root 对象调用 initialise（）函数完成的，但渲染并不直接作用在窗口中，而是在窗口中的视口（Viewport）里。接下来就要创建视口，代码如下：

```
virtual void createViewports(void)
    {
        // Create one viewport, entire window
        Viewport* vp = mWindow->addViewport(mCamera);
        vp->setBackgroundColour(ColourValue(0,0,0));
    }
```

由于 Viewport 是属于窗口的，所以通过 mWindow 指针调用 addViewport（）函数创建该窗口中的一个视口。视口是显示渲染结果的地方，而渲染结果是摄象机所“看”到的结果，所以也应该不难理解为什么创建 Viewport 需要用摄象机为参数。实际上引擎中的 Viewport 类中有一个数据成员就是 Camera 指针类型的。

视口创建完成后，调用 setBackgroundColour（）函数设置背景颜色。

- 创建场景

接下来需要向场景中加入一点内容了。由于 ExampleApplication.h 中的 ExampleApplication 类是 OGRE 应用框架的组成部分，所以在这里并不知道场景中有什么内容，createScene（）函数也就是一个纯虚函数，它需要将来开发人员在自己的 ExampleApplication 派生类中重载，以实现自己特定的场景。

```
virtual void createScene(void) = 0;        // pure virtual - this has to be overridden
```

需要注意的是：即便将来的应用程序选用了特殊的室外和室内场景管理器，也需要重载此函数来创建场景中的可移动物体，因为特殊场景管理器一般仅负责地形、房屋等不可移动的“世界”。

- 创建帧监听器

场景中的摄象机、操作者本人（对于第一人称应用来说）甚至其它物体是需要接收键盘、鼠标、游戏杆或 AI（人工智能）的控制以在场景中移动的，所以需要给这些控制提供一个访问场景中物体机会，帧监听器 ExampleFrameListener 就是完成这项功能的一个类。在 setup 函数的最后需要为 ExampleApplication 创建一个 ExampleFrameListener 的实例。代码如下：

```
virtual void createFrameListener(void)
    {
        mFrameListener= new ExampleFrameListener(mWindow, mCamera);
        mRoot->addFrameListener(mFrameListener);
    }
```

ExampleFrameListener 的实例被创建之后，被加入到 Root 对象中，关于 ExampleFrameListener 具体的工作方式和原理见后详述。

至此，setup（）函数结束，它已完成渲染之前的全部准备工作，接下来就由 Root 对象调用 startRendering（）函数指挥渲染系统开始连续的渲染过程。

ExampleFrameListener 类

ExampleFrameListener 的主要作用是监听系统输入，并对场景做出控制反应。对于非控制性的变化（如自动的动画），其状态更新也可以由 ExampleFrameListener 完成。

注意到 ExampleApplication 类中有一个数据成员 mFrameListener，它是 ExampleFrameListener 类的实例。ExampleFrameListener 类是 OGRE 应用框架的重要组成部分。它为开发人员提供控制场景中移动物体的能力。

从 ExampleFrameListener.h 的代码中可以看到 ExampleFrameListener 类派生自 FrameListener 类。FrameListener 类是在 OGRE 引擎 FrameListener.h 中定义的一个类。FrameListener 定义如下：

```
class _OgreExport FrameListener
{
public:
    // 帧渲染之前的事件处理方法
    virtual bool frameStarted(const FrameEvent& evt) { return true; }
    // 帧渲染之后的事件处理方法
    virtual bool frameEnded(const FrameEvent& evt) { return true; }
    virtual ~FrameListener() {}
};
```

OGRE 引擎在渲染过程中的每一帧之前调用 frameStarted 方法，而之后会调用 frameEnded 方法，通过这种方式，可以在渲染期间对场景物体（包括摄象机）进行移动、缩放等控制，当然也可以完成其它的处理。

注意到 frameStarted 和 frameEnded 都以 FrameEvent 为参数，在 FrameListener.h 中首先定义了 FrameEvent 结构，该结构定义如下：

```
struct FrameEvent
{
    /**上一次事件发生到当前时间点所过去的时间（秒），一个 fameStarted 到
    frameEnded 之间或一个 frameEnded 到 fameStarted 之间过去的时间。
    */
    Real timeSinceLastEvent;
    /** 两个同类型事件之间所过去的时间（秒），两个 frameStarted 之间或两个
    frameEnded 之间的时间，也就是一个完整帧所花去的时间。
    */
    Real timeSinceLastFrame;
};
```

该结构中定义的两个时间元素可以方便我们对帧事件的处理。

接下来我们开始看 ExampleFrameListener 类。

数据成员

```
// 事件处理器
EventProcessor* mEventProcessor;
```

```
// 输入设备
InputReader* mInputDevice;
// 摄象机
Camera* mCamera;
// 渲染窗口
RenderWindow* mWindow;
// 状态开关变量
bool mStatsOn;
// 是否使用缓冲输入方式
bool mUseBufferedInput;
// 截屏计数器
unsigned int mNumScreenShots;
```

成员函数

帧监听器对输入事件的处理方法有两种模式：立即模式和缓冲模式。一般来讲立即模式适合于 3D 场景漫游过程，当在每帧渲染之前，系统捕获输入设备状态，并根据这些状态对场景中的物体和摄象机进行控制。而缓冲模式适合于 GUI 界面的情况（如设置菜单），输入设备状态可以被发送到各 GUI 元素进行处理（如按钮被按下）。

所以，ExampleFrameListener 的构造函数有 useBufferedInput 参数，缺省情况是 false，也就是采用立即模式。此外，ExampleFrameListener 的构造函数还需要渲染窗口和当前摄象机为参数，因为 OGRE 支持多窗口渲染和多摄象机，ExampleFrameListener 需要知道控制的作用对象。

构造函数

```
ExampleFrameListener(RenderWindow* win, Camera* cam, bool useBufferedInput =
false)
{
    mUseBufferedInput = useBufferedInput;
    // 采用缓冲处理模式（适合于 GUI 界面的情况）
    if (mUseBufferedInput)
    {
        mEventProcessor = new EventProcessor();
        mEventProcessor->initialise(win);
        OverlayManager::getSingleton().createCursorOverlay();
        mEventProcessor->startProcessingEvents();
    }
    else
    {
        采用立即模式（适合与场景漫游的情况）
        mInputDevice = PlatformManager::getSingleton().createInputReader();
        mInputDevice->initialise(win,true, true);
```

```
        }
        mCamera = cam;
        mWindow = win;
        mStatsOn = true;
        mNumScreenShots = 0;
    }
```

构造函数针对不同的事件处理模式进行不同的初始化工作。对于缓冲处理模式，因为要对不同的 GUI 元素采取不同的事件处理策略（点击、拖动等），为便于实现，OGRE 采用一个事件处理器 EventProcessor 来进行事件处理。在 EventProcessor 类中封装了对不同事件的处理方式，见后详述。对于立即模式，OGRE 在 ExampleFrameListener 的构造函数中首先创建输入设备实例，而后初始化之。

如前所述，ExampleFrameListener 类是 FrameListener 类派生出来的具体类，而 FrameListener 中定义了 frameStarted 和 frameEnded 这两个将会在每一帧渲染前后被调用的纯虚函数，在这里 ExampleFrameListener 将实现这两个函数，以实现对场景物体的控制。一般情况下，仅需要 frameStarted 就足够了。

frameStarted 函数

```
    bool frameStarted(const FrameEvent& evt)
    {
        if (mUseBufferedInput)
        {
            // What to do here?
            return true;
        }
        else
        {
            return processUnbufferedInput(evt);
        }
    }
```

根据不同的事件处理模式，frameStarted 函数有不同的实现办法。对于缓冲模式，系统采用 EventProcessor 来进行事件处理，它本身也是 FrameListener 类的派生类，它有自己的事件处理办法，见后详述。而对于立即模式，系统调用 processUnbufferedInput 函数来对输入做出处理。

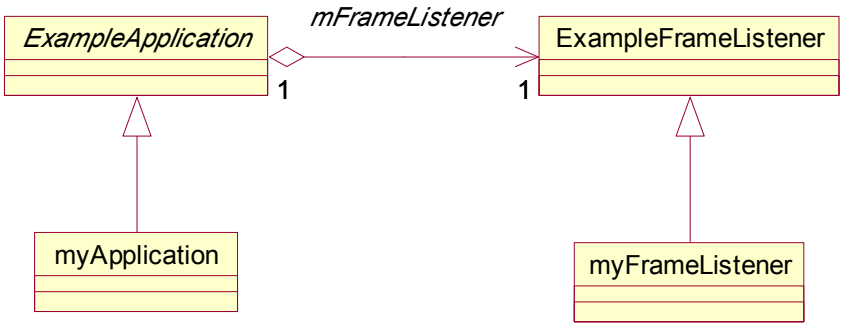
processUnbufferedInput 函数比较大，但它的主要功能是捕获输入设备状态，再根据输入设备情况对场景做相应处理。如：如果按下 ASWD 键则控制摄象机移动；如果鼠标移动则控制摄象机旋转，以观察四周；如果按下 OPIK 键则旋转整个场景；如果按下 F 键则控制 DEBUG 区域的显示与否；如果按下 PrintScreen 键则截取当前屏幕，并保存到硬盘上。

大家可能注意到，立即模式中 frameStarted 函数的主要功能是根据键盘和鼠标输入情况控制摄象机，其实你可以在实际应用的时候，在派生你自己的 myFrameListener 后，重新实现 frameStarted 函数，让其控制某些 3D 物体的自动动画、自动运动等。当然不要忘了还要让新的 frameStarted 函数调用基类 ExampleFrameListener 类中的 frameStarted 函数，不然你的键盘和鼠标控制功能就全部丢失了。

第一个 3D 程序

了解完 OGRE 的 FrameWork 后，我们开始使用该框架来进行应用程序的开发，大家会发现有了引擎和 FrameWork 后，3D 程序的开发会变得多么简单！

首先还是有必要回顾一下 FrameWork 和实际应用程序的关系。



FrameWork 与实际应用的关系

OGRE 的应用框架中定义好了 ExampleApplication 与 ExampleFrameListener 类，已封装了 3D 应用程序的全部要素，应用开发者所要做的工作主要有 2 点：

1、 派生出自己的应用程序类，重新实现 createScene 函数以创建场景。

2、 派生出自己的监听器类，如果需要的话，重新实现 frameStarted 函数以进行特殊的输入控制和动画控制。

打开 Demo_EnvMapping 工程，这是一个非常简单易懂的程序实例。

这个程序只有两个文件 EnvMapping.h 和 EnvMapping.cpp。

EnvMapping.h 文件定义了 ExampleApplication 类的派生类 EnvMapApplication，并重新实现了 createScene 函数创建出一个亮闪闪的具有环境帖图的食人魔头像。代码如下：

```
class EnvMapApplication : public ExampleApplication
{
public:
    EnvMapApplication() {}
protected:
    // 重新实现 createScene 函数，创建实际场景
    void createScene(void)
    {
        // 设置环境光
        mSceneMgr->setAmbientLight(ColourValue(0.5, 0.5, 0.5));
        // 创建点光源 l
        Light* l = mSceneMgr->createLight("MainLight");
        // 设置点光源 l 的位置，缺省颜色为白色
        l->setPosition(20,80,50);
        // 读入 ogrehead.mesh 模型文件，创建为一个 Entity。
        Entity *ent = mSceneMgr->createEntity("head", "ogrehead.mesh");
        // 设置食人魔 Entity 的材质为指定材质（环境贴图）
        ent->setMaterialName("Examples/EnvMappedRustySteel");
        // 将食人魔 Entity 连接到场景根节点上。
        mSceneMgr->getRootSceneNode()->createChild()->attachObject(ent);
```

```
    }
};
```

一个基本的场景包括各种光、摄象机和模型等。在这里没有显式创建摄象机的原因是采用了 ExampleApplication 类中创建的缺省摄象机（见 FrameWork 部分）。

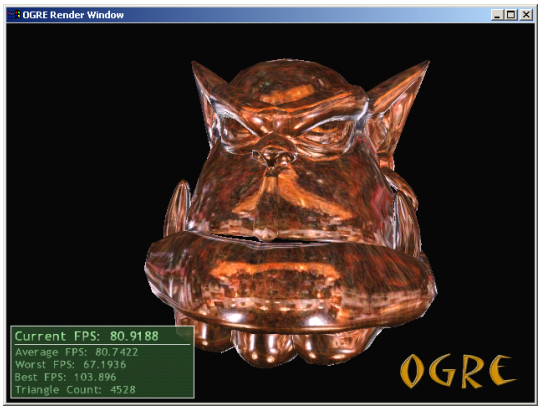
每个模型文件载入后被创建成 Entity。为了便于对场景的管理 OGRE 引入了场景节点的概念，所有的场景节点组合成一棵节点树，而全部 Entity 都挂在这棵节点树中的不同节点上。场景管理器 mSceneMgr 通过对节点树的操控来完成对场景物体的操控。关于这部分的内容参看后面的场景管理部分。

场景建立好之后，接下来处理应用程序的入口问题，这部分代码在 EnvMapping.cpp 里，代码如下：

```
// 判断编译平台，OGRE 是支持跨平台的。
#if OGRE_PLATFORM == PLATFORM_WIN32
    #define WIN32_LEAN_AND_MEAN
    #include "windows.h"
    // Win32 平台
    INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT )
#else
    // 非 Win32 平台
    int main(int argc, char **argv)
#endif
    {
        // 创建应用程序对象
        EnvMapApplication app;
        try {
                // 执行 go 函数，开始应用程序过程
                app.go();
            }
        // 异常处理
        catch( Exception& e )
                {
                // Win32 平台异常提示
                #if OGRE_PLATFORM == PLATFORM_WIN32
                    MessageBox( NULL, e.getFullDescription().c_str(), "An exception has occured!", MB_OK | MB_ICONERROR | MB_TASKMODAL);
                // 非 Win32 平台异常提示
                #else
                    fprintf(stderr, "An exception has occured: %s\n",
                    e.getFullDescription().c_str());
                #endif
                }
        return 0;
    }
```

编译本工程，生成 exe 文件，将该 exe 文件拷贝到运行环境中执行，即可看到结果。不要忘了 OGRE 程序的运行期环境（见前面运行期环境部分）和资源环境（模型文件、

纹理文件等）。



EnvMapping 程序

OGRE 中的数学

OGRE 是一个三维图形渲染引擎，在很多处理过程中都需要使用到数学这一基本手段，除了简单数学计算以外还需要用到大量的矩阵运算。在 OGRE 中提供了数学支撑环境，它们分别封装在 math、Vector3、Vector4、Matrix3、Matrix4 和 Quaternion 类里，对于特殊的需要极高效率的数学函数用汇编方式实现在 asm_math.h 文件中。

asm_math.h

在本头文件中针对不同的开发平台（MSVC 和 GNUC，但 GUNC 不支持嵌入汇编程序，所以对于 GNUC 其实还是调用其运行库函数）用汇编程序的方式实现了非常需要高效率的数学函数。其中包括 asm_arccos、asm_arcsin、asm_arctan、asm_sin、asm_cos、asm_tan、asm_sqrt、asm_rsq、apx_rsq、apx_rsq、InvSqrt、asm_rand、asm_rand_max、asm_ln 等函数。一般不需要由应用开发者直接调用这里定义的函数，这些函数常由 Math 类调用。

暂时不讲

Math 类

本类封装了基本的数学函数，大部分函数都是 C 运行库中相应函数的别名，使用这种方式的原因在于可以在这里提供额外的特性。 Math 类中的成员函数除构造和析构函数以外都是采用静态函数的方式提供的，其数据成员也都是静态成员。又由于 Math 类的实例化是由 Root 自动进行的，所以在使用的时候只需要以这样的格式调用数学函数就可以了：Math:: 函数名( )或 Math:: getSingleton( ).函数名（）

Math 类中的数学常量

在 Math 类中提供了部分使用频繁的常量。

POS_INFINITY：
定义如下：
const Real Math::POS_INFINITY = std::numeric_limits<Real>::infinity();
NEG_INFINITY：
定义如下：
const Real Math::NEG_INFINITY = -std::numeric_limits<Real>::infinity();
PI：
通过用汇编实现的 tan 函数（atan）计算出来的圆周率。
定义如下：
const Real Math::PI = Real(4.0 * atan(1.0));
TWO_PI：
定义如下：
const Real Math::TWO_PI = Real(2.0 * PI);
HALF_PI：
PI 的二分之一：
const Real Math::HALF_PI = Real(0.5 * PI);

数学函数速查表

取整数绝对值。
int Math::IAbs (int iValue)
取大于参数的最小整数。
int Math::ICeil (float fValue)
取小于参数的最大整数。
int Math::IFloor (float fValue)
取整数的符号，如果为正数取 1，为负数取-1，为 0 取 0。
int Math::ISign (int iValue)
取实数绝对值
Real Math::Abs (Real fValue)
求反余弦函数，结果在 0 到 PI 之间。实际上通过调用 asm_arccos 函数实现。
Real Math::ACos (Real fValue)
求反正弦函数，结果在-PI/2 到 PI/2 之间。实际上通过 asm_arcsin 函数实现。
Real Math::ASin (Real fValue)
求反正切函数，结果在-PI/2 到 PI/2 之间
Real Math::ATan (Real fValue)
求 2 个数之比的反正切值，实际通过 C 运行库函数实现
Real Math::ATan2 (Real fY, Real fX)
求大于参数的最小整数值，返回值为实数
Real Math::Ceil (Real fValue)
求根据弧度计算余弦值，第二个参数指定是否使用查表法。在实现上，如果使用查表法，将被转换为求加 90 度后的正弦。查表法的缺省精度为将圆分为 4096 份，忽略更小的差别。如果需要更改精度，给 Math 类的构造函数传参。如果不使用查表法，将转换为通过汇编实现的 asm_cos 函数求结果。

Real Math::Cos (Real fValue, bool useTables)

计算指数函数值，结果为 e 的 fValue 次方。e 为自然对数 2.7182818。通过 C 运行库的 exp 函数实现。

Real Math::Exp (Real fValue)

取小于参数的最大整数，结果为实数。

Real Math::Floor (Real fValue)

计算参数的自然对数值，实际通过汇编实现的 asm_ln 函数完成。

Real Math::Log (Real fValue)

计算某数的某次方，实际通过 C 运行库的 pow 函数实现。

Real Math::Pow (Real fBase, Real fExponent)

取实数的符号，大于 0 返回 1.0，小于 0 返回-1.0，等于 0 返回 0。

Real Math::Sign (Real fValue)

求正弦函数，第二个参数指定是否使用查表法，缺省值为 false。如果不查表，就通过汇编实现的 asm_sin 函数完成计算。

Real Math::Sin (Real fValue, bool useTables)

计算实数的平方

Real Math::Sqr (Real fValue)

计算实数的开平方，实际通过汇编实现的 asm_sqrt 函数完成计算。

Real Math::Sqrt (Real fValue)

计算实数开平方后的倒数，例如：1 / Sqrt(x)，对于向量归一化很有好处。

Real Math::InvSqrt (Real fValue)

返回 0、1 之间的随机数，包括 0 和 1。通过汇编实现的 asm_rand 和 asm_rand_max 函数完成。

Real Math::UnitRandom ()

返回两个指定参数之间的随机数。

Real Math::RangeRandom (Real fLow, Real fHigh)

返回-1、1 之间的随机数，包括-1 和 1。

Real Math::SymmetricRandom ()

角度转换为弧度

Real Math::DegreesToRadians(Real degrees)

弧度转变为角度

Real Math::RadiansToDegrees(Real radians)

在二维空间里判断一个点是否在一个三角形里，如果是返回真，否则返回假。

bool Math::pointInTri2D(Real px, Real py, Real ax, Real ay, Real bx, Real by, Real cx, Real cy)

求正切函数，第二个参数指定是否使用查表法，缺省值为 false。如果不查表，就通过汇编实现的 asm_tan 函数完成计算。

Real Math::Tan(Real radians, bool useTables)

判断两个值是否相等，第三个参数是允许的误差度，例如：a=5、b=6、tolerance=2，则结果是 a=b。但 tolerance 的缺省值为 std::numeric_limits<Real>::epsilon()。

bool Math::RealEqual(Real a, Real b, Real tolerance)

Vector3 类

表达三维向量 V（x，y，z）的类，三维世界中的位置、方向和缩放因子都可以用 Vector3 来表达，关键看你如何解释与使用它。

为了提高效率，Vector3 类的成员函数大部分都实现为内联函数。又为了将来操作方便，Vector3 类的数据成员都实现为 public 类型。

常量

```
static const Vector3 ZERO;
static const Vector3 UNIT_X;
static const Vector3 UNIT_Y;
static const Vector3 UNIT_Z;
static const Vector3 UNIT_SCALE;
```

分别代表零向量、X 轴单位向量、Y 轴单位向量、Z 轴单位向量和单位缩放因子（其实是不缩放），这些向量使用频繁，所以实现为常量。

成员函数

重载[ ]操作符，通过下标 1，2，3 可以取 x，y，z 分量。

```
inline Real operator [ ] ( unsigned i ) const
```

重载[ ]操作符，返回引用，可以做左值。

```
inline Real& operator [] ( unsigned i )
```

重载赋值操作符，返回引用，结果可以做左值。

```
inline Vector3& operator = ( const Vector3& rkVector )
```

重载等号操作符，判断两个向量是否相等。

```
inline bool operator == ( const Vector3& rkVector ) const
```

重载不等号操作符，判断两个向量是否不相等。

```
inline bool operator != ( const Vector3& rkVector ) const
```

重载加号操作符，完成向量相加。

```
inline Vector3 operator + ( const Vector3& rkVector ) const
```

重载乘法操作符，完成向量与一个数的乘法。

```
inline Vector3 operator * ( Real fScalar ) const
```

重载乘法操作符，完成两个向量的相乘。

```
inline Vector3 operator * ( const Vector3& rhs) const
```

重载除法操作符，完成向量除以一个数的运算。

```
inline Vector3 operator / ( Real fScalar ) const
```

重载负号操作符，将原向量的每个分量取负。

```
inline Vector3 operator - () const
```

重载乘法操作符，用友员函数的方式实现，实现一个向量与一个数相乘。

```
inline friend Vector3 operator * ( Real fScalar, const Vector3& rkVector )
```

重载+=运算符，A+=B。

inline Vector3& operator += (const Vector3& rkVector)

重载-=运算符，A-=B。

inline Vector3& operator -= (const Vector3& rkVector)

重载*=运算符，A*=B。

inline Vector3& operator *= (Real fScalar)

重载/=运算符，A/=B。

inline Vector3& operator /= (Real fScalar)

求向量长度，因为求结果用到开方，会消耗 CPU 时间，所以如果有可能就使用后面的求向量长度平方的函数。

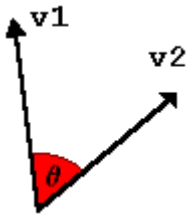
inline Real length () const

求向量长度的平方，很多时候用它可以代替求向量长度的函数，因为没有开平方，所以可以提高效率。

inline Real squaredLength () const

计算向量点积，向量点积有如下应用：

Cos(theta) = DotProduct(v1,v2) / (length(v1) * length(v2))



所以，当知道两个向量的点积和长度之后就可以获得向量之间的夹角的余弦。如果两个向量都是单位向量的化，则点积的结果就是夹角的余弦。

inline Real dotProduct(const Vector3& vec) const

向量归一化，转变为方向不变的单位向量。归一化的方法是每个分量都除以向量的长度。

inline void normalise(void)

求向量叉积。

向量叉积的结果代表这两个向量所在平面的法向量。

向量叉积遵循右手法则，从第一向量握向第二向量，大拇指指向的向量就是叉积结果。例如：屏幕左边线为第一向量，屏幕下边线为第二向量，那么它们的叉积结果就指向屏幕里面。向量的叉积计算可以用来判断两个向量的拓扑关系（左右之分）。

inline Vector3 crossProduct(const Vector3& rkVector) const

向量可以表达点的坐标，本函数的功能是计算两个点之间的中点坐标。

inline Vector3 midPoint(const Vector3& vec) const

重载小于操作符，如果 A 向量的各分量都小于 B 向量，则 A 向量小于 B 向量。

inline bool operator < (const Vector3& rhs) const

重载大于操作符，如果 A 向量的各分量都大于 B 向量，则 A 向量大于 B 向量。

inline bool operator > (const Vector3& rhs) const

将 A、B 两向量中的各分量的小值组合成新的向量。

inline void makeFloor(const Vector3& cmp)

将 A、B 两向量中的各分量的大值组合成新的向量。

inline void makeCeil(const Vector3& cmp)

计算并返回一个向量的垂直向量，一个向量的垂直向量有无穷多个，这里只返回一个。

用原向量与X正轴向量做叉乘得到的结果向量是原向量与X正轴向量构成的平面的法向量，肯定垂直于原向量，函数返回这个结果向量。如果发现得到的结果向量的长度为0，又因为Length(VectorC) = Length(Vector1) * Length(Vector2) * sin(theta)，证明原向量与X正轴重合（sin(theta)为0），那么就改为求原向量与Y正轴叉乘作为结果向量。

```
inline Vector3 perpendicular(void)
```

得到偏离原向量一定角度的随机向量。

本函数要求随机数种子已设定好。当然Math类的构造函数设定了随机数种子。如果在调用本函数之前，初始化了Math类，就可以正常使用本函数。

第一个参数是允许偏移的最大角度。

第二个参数是向量，有缺省值。要求是原向量的一个垂直向量。

```
inline Vector3 randomDeviant(
    Real angle,
    const Vector3& up = Vector3::ZERO )
```

得到能将当前向量用最小路径旋转到目的向量的四元组。

```
Quaternion getRotationTo(const Vector3& dest) const
```

重载输出流操作符。

```
inline _OgreExport friend std::ostream& operator <<
    ( std::ostream& o, const Vector3& v )
```

Vector4 类

包含x, y, z, w四个分量的向量。

构造函数

缺省构造函数
inline Vector4()
接收四个实数的构造函数
inline Vector4(Real fX, Real fY, Real fZ, Real fW)
接收四个元素的实数数组的构造函数
inline Vector4(Real afCoordinate[4])
接受四个int的数组的构造函数
inline Vector4(int afCoordinate[4])
接受实数指针的构造函数
inline Vector4(const Real* const r)
拷贝构造函数
inline Vector4(const Vector4& rkVector)

操作符重载

inline Real operator [] (unsigned i) const

inline Real& operator [] (unsigned i)
inline Vector4& operator = (const Vector4& rkVector)
inline bool operator == (const Vector4& rkVector) const
inline bool operator != (const Vector4& rkVector) const
用 Vector3 对 Vector4 赋值，其中 w 为 1。
inline Vector4& operator = (const Vector3& rhs)
Vector4 乘以 Matrix4
inline Vector4 operator * (const Matrix4& mat)
重载输出流操作符。
inline _OgreExport friend std::ostream& operator <<(std::ostream& o, const Vector4& v)

Matrix3 类

3*3 矩阵类。

常量

const Real Matrix3::EPSILON = 1e-06;
const Matrix3 Matrix3::ZERO(0,0,0,0,0,0,0,0,0);
const Matrix3 Matrix3::IDENTITY(1,0,0,0,1,0,0,0,1);
const Real Matrix3::ms_fSvdEpsilon = 1e-04;
const int Matrix3::ms_iSvdMaxIterations = 32;

构造函数

```
Matrix3 ();
Matrix3 (const Real arr[3][3]);
Matrix3 (const Matrix3& rkMatrix);
Matrix3 (Real fEntry00, Real fEntry01, Real fEntry02,
                Real fEntry10, Real fEntry11, Real fEntry12,
                Real fEntry20, Real fEntry21, Real fEntry22);
```

操作符重载

下标操作符
Real* operator[] (int iRow) const;
指针操作符
operator Real* ();
Matrix3& operator= (const Matrix3& rkMatrix);
bool operator== (const Matrix3& rkMatrix) const;
bool operator!= (const Matrix3& rkMatrix) const;
Matrix3 operator+ (const Matrix3& rkMatrix) const;

Matrix3 operator- (const Matrix3& rkMatrix) const;

Matrix3 operator* (const Matrix3& rkMatrix) const;

负号操作符

Matrix3 operator- () const;

Matrix3 乘 Vector3，结果是 3*1 的 Vector3。

Vector3 operator* (const Vector3& rkVector) const;

以友员方式重载的乘号操作符，Vector3 乘 Matrix3，结果是 1*3 的 Vector3。

friend Vector3 operator* (const Vector3& rkVector,const Matrix3& rkMatrix);

Matrix3 乘以一个标量。

Matrix3 operator* (Real fScalar) const;

友员方式重载的标量乘 Matrix3。

friend Matrix3 operator* (Real fScalar, const Matrix3& rkMatrix);

取得 Matrix3 中的指定列。

Vector3 GetColumn (int iCol) const;

设定 Matrix3 中的指定列。

void SetColumn(int iCol, const Vector3& vec);

指定三个轴向量，构成一个 Matrix3,注意：每个轴在 Matrix3 表现为一列。

void Matrix3::FromAxes(const Vector3& xAxis, const Vector3& yAxis, const Vector3& zAxis)

求当前矩阵的转置

Matrix3 Transpose () const;

求当前矩阵的逆矩阵，注意返回值是 bool 型的，得到的结果矩阵在第一个参数里。

bool Matrix3::Inverse (Matrix3& rkInverse, Real fTolerance) const

求当前矩阵的逆矩阵，返回值是结果矩阵。

Matrix3 Matrix3::Inverse (Real fTolerance) const

求本矩阵代表行列式的值

Real Matrix3::Determinant () const

还有一些成员函数功能不明，且未被使用过。

Matrix4 类

4*4 矩阵。

常量

Matrix4::ZERO 零矩阵

Matrix4::IDENTITY 单位矩阵

构造函数：

```
inline Matrix4()
inline Matrix4(Real m00, Real m01, Real m02, Real m03,
               Real m10, Real m11, Real m12, Real m13,
               Real m20, Real m21, Real m22, Real m23,
               Real m30, Real m31, Real m32, Real m33 )
```

成员函数：

重载下标操作符
inline Real* operator [] (unsigned iRow)
重载下标操作符
inline const Real *const operator [] (unsigned iRow) const
矩阵乘法
inline Matrix4 concatenate(const Matrix4 &m2)
矩阵乘法
inline Matrix4 operator * (const Matrix4 &m2)
用当前矩阵完成 Vector3 的变换，返回值是结果 Vector3。
inline Vector3 operator * (const Vector3 &v) const
矩阵加法
inline Matrix4 operator + (const Matrix4 &m2) const
矩阵减法
inline Matrix4 operator - (const Matrix4 &m2)
重载等号
inline bool operator == (const Matrix4& m2) const
重载不等号
inline bool operator != (Matrix4& m2) const
将 Matrix3 转换为 Matrix4
inline void operator = (const Matrix3& mat3)
矩阵转置
inline Matrix4 transpose(void)
设置变换矩阵中的平移部分。
结果是：

原值	原值	原值	v.x
原值	原值	原值	v.y
原值	原值	原值	v.z
原值	原值	原值	原值

inline void setTrans(const Vector3& v)
构建平移变换矩阵
得到的结果是：

0	0	0	v.x
0	0	0	v.y
0	0	0	v.z
0	0	0	0

inline void makeTrans(const Vector3& v)
构建平移变换矩阵
结果是：

0	0	0	tx
0	0	0	ty
0	0	0	tz
0	0	0	0

inline void makeTrans(Real tx, Real ty, Real tz)
由 Vector3 获得一个平移变换矩阵。注意是静态成员函数。
结果是：

0	0	0	v.x
0	0	0	v.y
0	0	0	v.z
0	0	0	0

inline static Matrix4 getTrans(const Vector3& v)
由 3 个实数获得一个平移变换矩阵。注意是静态成员函数。
结果是：

0	0	0	t_x
0	0	0	t_y
0	0	0	t_z
0	0	0	0

inline static Matrix4 getTrans(Real t_x, Real t_y, Real t_z)
设置矩阵中的缩放部分
结果是：

v.x	原值	原值	原值
原值	v.y	原值	原值

原值	原值	v.z	原值
原值	原值	原值	原值

inline void setScale(const Vector3& v)
由 Vector3 获得缩放矩阵
结果是：

v.x	0	0	0
0	v.y	0	0
0	0	v.z	0
0	0	0	0

inline static Matrix4 getScale(const Vector3& v)
由三个实数值获得缩放矩阵
结果是：

s_x	0	0	0
0	s_y	0	0
0	0	s_z	0
0	0	0	0

inline static Matrix4 getScale(Real s_x, Real s_y, Real s_z)
从 4*4 矩阵中析出旋转和缩放矩阵，参数是引用方式传参，接收结果。
inline void extract3x3Matrix(Matrix3& m3x3)
重载输出流操作符
inline _OgreExport friend std::ostream& operator << (std::ostream& o, const Matrix4& m)

Quaternion 类

四元数最早是为了扩展复数应用而产生与发展起来的。然而，人们发现四元数也可以应用在计算机图形学上，作为表现旋转的可选择方法之一。

普通的复数形式可以写成 xi+y，其中 x 是虚部，y 是实部。不可思议的是 i 的平方等于 -1，那个 -1 的平方根就像你应该知道的那样，其并不存在。因此取名叫“虚数”。

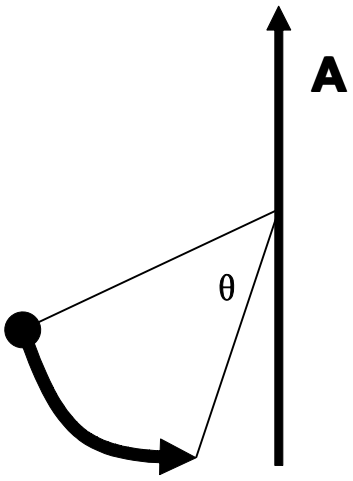
假如 xi + y 是一个复数，那么我们可认为四元数是一组很特别的复数。四元数不只一个虚部，而是有三个。四元数的形式可写成 xi + yj + zk + w，在这里 i、j、k 的平方都等于-1。从现在起，让我们忘记四元数中平方根的形式，其就可写成这样[w,(x y z)]。

四元数可用来表现所有形式的旋转。正如你所看到，四元数仅需要四个浮点数，所以它只是给对应的矩阵添加了第四个数而已。这个属性可有助于将你许多的旋转处理存储到文件中去（例如关键帧动画）。但你如何获取源于[w,(x y z)]的旋转呢？

答案在这，你首先需要计算你的四元数的模。四元数的模与矢量的有点相似。它可做 $sqrt(w^2 + x^2 + y^2 + z^2)$的计算。假如平均值为 1，你就得到所谓的单位四元数（正象单位矢量）。

唯一的是四元数需要表现旋转，因此你必须对现有的四元数进行归一化。

四元数并不天生就具有旋转的特性，因此你还要做一些转换四元数或其它的一些必要工作。你可将存储空间的四元数转换为欧拉角或轴/角的表示法，及转换为可供 OpenGL 使用的矩阵。我们平常描述某个点饶某个轴旋转往往是用轴/角形式描述，例如：点 P 饶 A 轴旋转θ 度。那么描述这个旋转的四元数就定义为：Q = （c，s X_A，s Y_A，s Z_A）。其中 s = sin(θ/2)，c = cos(θ/2)。



静态数据成员

零四元数
Quaternion Quaternion::ZERO(0.0,0.0,0.0,0.0);
单位四元数
Quaternion Quaternion::IDENTITY(1.0,0.0,0.0,0.0);

构造函数

带四个参数的构造函数，为 w，x，y，z 赋值。
Quaternion::Quaternion (Real fW, Real fX, Real fY, Real fZ)
拷贝构造函数
Quaternion::Quaternion (const Quaternion& rkQ)

成员函数

由 3*3 旋转矩阵生成四元数，参数是旋转矩阵。
void Quaternion::FromRotationMatrix (const Matrix3& kRot)
将四元数转换为旋转矩阵，参数是回传参数，接收得到的旋转矩阵。
void Quaternion::ToRotationMatrix (Matrix3& kRot) const
从角、轴生成四元数，参数为角和轴，即饶什么轴旋转多少度。

void Quaternion::FromAngleAxis (const Real& rfAngle,

由四元数生成轴、角表示法的角和轴。参数是引用参数，回传用途。

void Quaternion::ToAngleAxis (Real& rfAngle, Vector3& rkAxis) const

由 3 个向量表达的旋转矩阵生成四元数，参数是指针指向的多个向量表达的旋转矩阵（一般为 3 个）

void Quaternion::FromAxes (const Vector3* akAxis)

由 3 个向量表达的旋转矩阵生成四元数

void FromAxes (const Vector3& xAxis, const Vector3& yAxis, const Vector3& zAxis);

由四元数生成旋转矩阵，该矩阵由指针指向的多个（3 个）向量表达。

void ToAxes (Vector3* akAxis) const;

由四元数生成旋转矩阵，该矩阵由 3 个向量表达。

void ToAxes (Vector3& xAxis, Vector3& yAxis, Vector3& zAxis);

重载=操作符

Quaternion& operator= (const Quaternion& rkQ);

重载+操作符

Quaternion operator+ (const Quaternion& rkQ) const;

重载-操作符

Quaternion operator- (const Quaternion& rkQ) const;

重载*操作符，四元数*四元数。

Quaternion operator* (const Quaternion& rkQ) const;

重载*操作符，四元数*标量。

Quaternion operator* (Real fScalar) const;

重载*操作符，四元数*四元数。重载为友员方式。

friend Quaternion operator* (Real fScalar, const Quaternion& rkQ);

重载负号操作符

Quaternion operator- () const;

重载==操作符

bool operator== (const Quaternion& rhs) const;

四元数点乘

Real Dot (const Quaternion& rkQ) const;

四元数求模

Real Norm () const;

四元数求倒数，要求本四元数非 0

Quaternion Inverse () const;

单位四元数求倒数，要求本四元数为单位四元数。

Quaternion UnitInverse () const; // apply to unit-length quaternion

求四元数的指数，即 e 的四元数次方

Quaternion Exp () const;

以 e 为底求四元数的对数

Quaternion Log () const;

用四元数旋转向量（似乎 nVidia 的 SDK 才支持）。

Vector3 operator* (const Vector3& rkVector) const;

静态成员函数，球面线性插值。参数 fT 为 0-1 的时间因子。

static Quaternion Slerp (Real fT, const Quaternion& rkP, const Quaternion& rkQ);

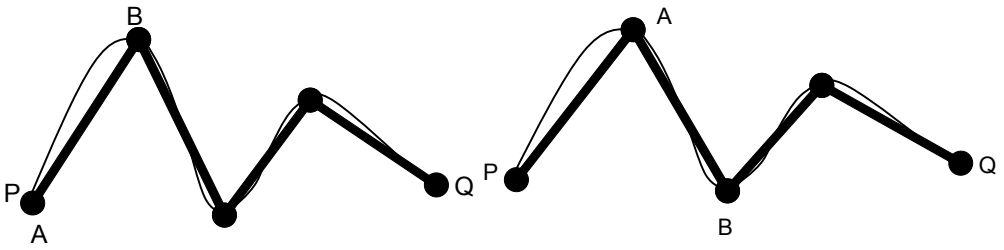
静态成员函数，带干扰因子的球面线性插值。参数 fT 为 0-1 的时间因子。

static Quaternion SlerpExtraSpins (Real fT, const Quaternion& rkP, const Quaternion& rkQ,
int iExtraSpins);

静态成员函数，目前不清楚用途。

static void Intermediate (const Quaternion& rkQ0,
const Quaternion& rkQ1, const Quaternion& rkQ2,
Quaternion& rka, Quaternion& rkB);

静态成员函数，样条插值（二次插值）。目的是在已知样条形状的情况下，通过插值得到平滑的目的线条。fT 为 0-1 的时间因子。rkP 和 rkQ 为样条的起点和终点，而 rkA 和为 rkB 插值过程中的某一线段的起点和终点（控制点连线）。



对 AB 段插值　　对新的 AB 段插值

static Quaternion Squad (Real fT, const Quaternion& rkP,
const Quaternion& rkA, const Quaternion& rkB,
const Quaternion& rkQ);

重载输出流操作符。

inline _OgreExport friend std::ostream& operator <<
(std::ostream& o, const Quaternion& q)

异常处理和日志

日志管理

用文件来记录 Ogre 系统初始化、运行、结束以及调试信息。使用日志便于我们调试程序。

Ogre 日志系统的组成

Ogre 日志系统由两个类组成：Log 类与 LogManager。下面我们分别来看这两个类。

Log 类

代表用于记录信息的日志。Log 类的一个对象对应于一个日志文件。

Log 类提供了向日志文件写信息的函数 logMessage，其定义如下：

```
void logMessage(const String& message, LogMessageLevel lml = LML_NORMAL);
```

参数 message，String 类型的变量，存储要写入的信息。

参数 lml，指定传入信息的级别，来衡量一条信息的重要程度。

参数值	信息的重要程度
LML_TRIVIAL	最低
LML_NORMAL	一般
LML_CRITIAL	最高

为了判断一条信息的重要性，从而决定是否将该信息写入日志文件。我们不光要衡量信息的重要程度，还应该同时考虑日志文件的重要程度。Log 类提供了一个函数 setLogDetail 来设置日志文件的重要程度，其定义如下：

```
void setLogDetail(LoggingLevel ll);
```

参数 ll，指定该日志文件的级别，来衡量日志文件的重要程度。

参数值	代表日志文件的重要程度
LL_LOW	最低
LL_NORMAL	一般
LL_BOREME	最高

LogManager 类

管理所有 Log 类的对象，也就是管理所有的日志文件。并负责向日志文件中输出信息。

LogManager 类提供了创建 Log 对象的成员函数 createLog。

注意，不要用 Log 类直接创建对象，而要用 LogManager 的 createLog 函数来创建 Log 对象。因为这保证了：使 LogManager 维护所有的 Log 对象，通过 LogManager 可以方便地进行查找等操作。createLog 定义如下：

```
Log* createLog( const String& name, bool defaultLog = false, bool debuggerOutput = true );
```

参数name是String类型的，它指定所创建日志文件的文件名，如Ogre.log。

参数defaultLog是布尔类型的，如果为true，则把当前创建的日志文件设置为LogManager默认的日志文件。调用LogManager的接口函数都对此文件生效。

参数debuggerOutput是布尔类型的，如果为true，则不只向日志文件中输出信息，还向调试窗口中输出信息。

成员函数

成员函数getLog可以通过文件名得到其代表的Log对象。它的定义如下：

```
Log* getLog( const String& name);
```

参数name指定了要查找的日志文件名。

设置默认Log文件级别的成员函数：

```
void setLogDetail(LoggingLevel ll);
```

参数ll指定默认日志文件的重要程度，它可以是：

参数值	信息的重要程度
LML_TRIVIAL	最不重要
LML_NORMAL	一般
LML_CRITIAL	最重要

成员函数getDefaultLog返回默认的Log对象

```
Log* getDefaultLog();
```

成员函数logMessage向默认的日志文件中写入信息，它有两种形式：

```
void logMessage( const String& message, LogMessageLevel lml = LML_NORMAL);
void logMessage( LogMessageLevel lml, const char* szMessage, ... );
```

参数lml指定传入的信息的重要程度。

第一个logMessage可以向日志文件中写入一条信息。第二个logMessage可以写入接收的多条信息。

Ogre 日志系统的使用举例

程序概述

重载 ExampleApplication 内的 createScene 函数，在函数内创建一个名为 Test.log 的日志文件，并向该文件中写入一段信息。

部分代码

```
void createScene(void)
{
    Log *p_Log = LogManager::getSingleton().createLog( "test.log" );
    p_Log->logMessage( "I write a message to test.log" );
}
```

在createScene函数中，首先调用LogManager的createLog成员函数来创建一个名为test.log的日志文件，并用p_Log保存此日志文件的指针。然后，我们通过Log的成员函数logMessage向test.log中写入一句话。

执行一遍程序后，你会发现程序可执行文件所在的目录下多了一个名为test.log的文件，接着打开test.log，你会看见一行字：I write a message to test.log。
当然，这个程序所做的并没有什么意义。它的目的只是演示一下Ogre日志系统的使用方法，

关于Ogre日志系统的应用，在下一部分异常处理中还有介绍。

异常处理

Ogre 用 C＋＋内建的异常处理机制来处理错误。

使用异常处理的好处

首先，使用异常处理，可以使 C＋＋程序的两个部分相互通信（这两部分通常是分别开发的）。检测到异常的部分可以抛出异常，而另一部分可以捕获异常并做进行处理。

其次，使用异常处理，可以避免用返回值来定义错误。当有错误发生时，一个异常就会被抛出，这个异常里面封装了发生错误的详细信息。

Ogre 中使用异常的例子：

```
try {
    app.go();
} catch( Ogre::Exception& e ) {
    // 处理异常
}
```

首先，这段代码分为两部分，分别由 try 与 catch 包围起来，如果 app.go()有错误产生，则 Ogre::Exception 类型的异常会被抛出，catch 部分捕获并处理这个异常。

其次，如果没有异常，这段代码可能会是这样的：

```
if( !app.go() )
{
    // 处理错误
}
```

我们不得不用函数的返回值来定义错误。这样，不但错误信息不准确，而且导致程序的错误处理代码与应用代码混在一起，不易于维护。

Ogre 对异常处理的支持

Ogre::Exception

Ogre 定义了自己的异常类型 Ogre::Exception，它记录了错误的详细信息(错误编号、详细描述、错误发生的文件名、行数等)。当有错误发生时，Ogre 会抛出这个类型的异常，并把这个异常记录的错误信息写入到 LogManager 的默认日志文件中。

Ogre::Exception 提供了一个成员函数 getFullDescription，它的返回值是 String 类型的，保存了对错误的详细描述。

Ogre::Exception 维护了一个函数名称堆栈。提供了一个入栈函数_pushFunction，它接收一个 String 类型的参数。和一个出栈函数_popFunction。我们需要在每个函数的起始位置调用_pushFunction，并把该函数的函数名当作参数传入。在结束位置调用_popFunction。

几个辅助宏

Except(num, desc, src) 相当于 throw(Exception(num, desc, src, __FILE__, __LINE__))。a, b, c 分别代表错误编号，错误描述与错误发生所在的函数。__FILE__和__LINE__是系统变量，它们的意思是错误发生时的代码文件和行数。

OgreGuard(a) 相当于 Exception::_pushFunction((a))

OgreUnguard(a) 相当于 Exception::_popFunction((a))

Ogre 异常处理的例子

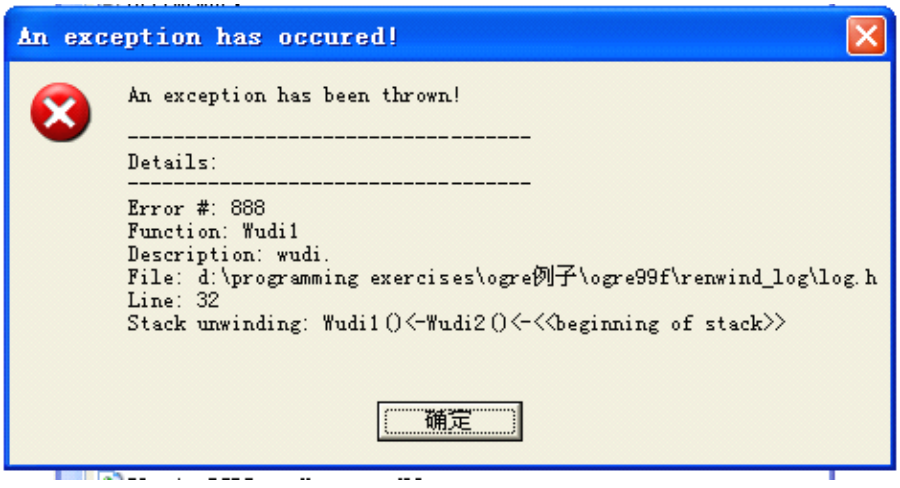
部分代码：

```
// Log.cpp
    MaterialApplication app;

try {
    app.go();
} catch( Exception& e ) {
#if OGRE_PLATFORM == PLATFORM_WIN32
    // 如果是 Win32 平台，则弹出一个错误对话框，显示错误的详细信息。
    MessageBox( NULL, e.getFullDescription().c_str(), "An exception has occured!", MB_OK |
                MB_ICONERROR | MB_TASKMODAL);
#else
    // 如果不是 Win32 平台，则直接输出错误的详细信息。
    fprintf(stderr, "An exception has occured: %s\n",
            e.getFullDescription().c_str());

// Log.h
// 定义一个函数 Wudi1，在内部武断地抛出一个异常
void Wudi1()
{
    OgreGuard( "Wudi1" );              // 将函数名称压入堆栈
    Except( 888, "wudi", "Wudi1" );    // 抛出一个异常
    OgreUnguard();                     // 弹出栈顶元素
}
```

```
// 定义一个函数 Wudi2，在内部调用 Wudi1
void Wudi2()
{
    OgreGuard( "Wudi2" );          // 将函数名称压入堆栈
    Wudi1();                       // 调用 Wudi1
    OgreUnguard();                 // 弹出栈顶元素
}

class MaterialApplication : public ExampleApplication
{
public:
    MaterialApplication() {}

protected:
    // 重载 createScene 函数
    void createScene(void)
    {
        Wudi2();            // 调用 Wudi2 函数
    }
};
```

执行结果

运行程序，会弹出一个错误对话框：



在默认的日志文件 Ogre.log 中也记录了同样的错误信息：

```
5:49:48: An exception has been thrown!

---------------------------------
Details:
---------------------------------
```

Error #: 888
Function: Wudi1
Description: wudi.
File: d:\programming exercises\ogre 例子\ogre99f\renwind_log\log.h
Line: 32
Stack unwinding: Wudi1()<-Wudi2()<-<<beginning of stack>>

最后一行，显示了函数名称队列的展开。我们可以看出，栈底的函数是 Wudi2()，异常发生在栈顶 Wudi1()函数内。

通过 Ogre 的日志系统与异常处理，我们可以迅速的查看系统运行的信息，快速定位错误发生的位置，并动态地处理错误。

场景结构体系

3D 场景是一个 3D 程序的实质性的内容。如果 3D 物体及各种光被胡乱堆砌在场景里，那么场景也就不是场景而是垃圾场了。场景需要有序的组织，这是任何一个 3D 引擎都必须解决的问题。

OGRE 为了解决场景管理的问题提出了几个重要的概念并将它们实现为引擎中的类：

Entity：场景元素，Mesh（模型）在场景中的实例。

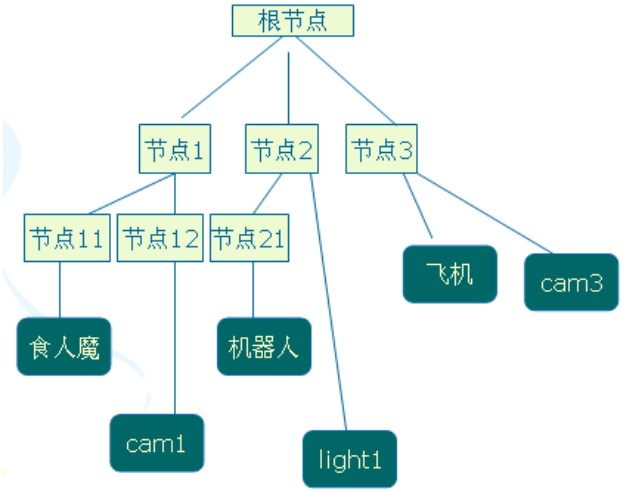
Light：场景元素，现实世界的光在场景中的实例。

Camera：场景元素，现实世界的观察者在场景中的实例。

SceneNode：抽象的场景管理单元。

注意到 SceneNode 和其它几个概念不一样，其它几个概念都是现实事物的程序抽象，而 SceneNode 在现实世界中并不存在，它是为了组织场景而提出的抽象概念。

OGRE 的思想是这样的：将现实世界中的场景划分成抽象的不同空间，区域中还可以划分成不同的小空间，每个空间由一个 SceneNode 对象来管理，SceneNode 将处理移动、旋转和缩放等与空间相关的行为。在每个 SceneNode 上可以挂接各种场景元素（如：Entity 、Light、Camera 等），场景元素本身并不负责与空间位置相关的行为，全部交给 SceneNode 来做（Light 和 Camera 类也保存自己的位置，但很多时候也都挂接到 SceneNode 上来处理，见后详述）。 OGRE 将大量 SceneNode 按照空间的划分层次组织成树状结构，从而完成对整个场景的有序组织。场景结构图如下：



场景结构图

根节点的位置往往在场景的中心。每个节点对象都用数据成员保存相对于父节点的相对位移和相对旋转量（这一点很重要）。节点类提供节点的移动和旋转函数。这样当节点 1 发生移动和旋转的时候，其全部子节点及它们挂接的场景元素也将被动地移动和旋转。想象一下 5 个人和一条狗坐在汽车里的情况，当车转弯的时候，5 个人和狗的相对位置虽然没有变，但他们都随着汽车转动了！这就是用节点树组织场景的巨大威力。再想一下，当缩小并旋转根节点的时候，整个世界都缩小和旋转起来。

OGRE 提供一个 SceneManager 类来负责整个场景节点树的维护。SceneManager 还负责 Entity、Light 和 Camera 的管理和维护。所以我们在对场景进行操作的时候更多的是与 SceneManager 打交道。

Entity 类

Mesh 模型在场景中的实例。

重要函数

void setMaterialName(const String& name);

参数为材质脚本文件（还记得 OGRE 运行环境中扩展名为.material 的脚本文件吗？）中定义的某个材质名称。将该材质指定为本 Entity 的材质。

注意：我们几乎不在程序中直接创建一个 Entity，对场景中的 Entity 的创建和管理一般是由 SceneManager 来完成的，所以我们都通过调用 SceneManager 的 createEntity 函数来创建 Entity。

SceneNode 类

场景节点类。

重要函数

从当前节点对象创建子节点。系统自动为子节点命名。参数是子节点相对于当前节点的位移和旋转量。

SceneNode* createChild(const Vector3& translate = Vector3::ZERO, const Quaternion& rotate = Quaternion::IDENTITY);

从当前节点对象创建子节点。第一个参数是该子节点的名称，其余参数是子节点相对于当前节点的位移和旋转量。

SceneNode* createChild(const String& name, const Vector3& translate = Vector3::ZERO, const Quaternion& rotate = Quaternion::IDENTITY);

按索引号获取子节点

SceneNode* getChild(unsigned short index) const;

按名称获取子节点

SceneNode* getChild(const String& name) const;

脱钩指定索引号的子节点

SceneNode* removeChild(unsigned short index);

脱钩指定名称的子节点

SceneNode* removeChild(const String& name);

在当前节点下挂接 MovableObject。MovableObject 是 Entity、Light、Camera 类的基类。

void attachObject(MovableObject* obj);

在当前节点下挂接光，在内部委托 attachObject 完成。

void attachLight(Light* l);

在当前节点下挂接摄象机，在内部委托 attachObject 完成。

void attachCamera(Camera* ent);

脱钩指定索引号的 MovableObject

MovableObject* detachObject(unsigned short index);

脱钩指定名称的 MovableObject

MovableObject* detachObject(const String& name);

还有几个函数没写完，从 Node 类继承而来的大量函数，如 setScle( )等

SceneManager 类

场景管理类，是场景管理的核心类。

SceneManager 类内部保存 std::map 类型的 CameraList、LightList、EntityList 和 SceneNodeList，并提供各种方法对它们进行管理和维护。SceneNode 自身的设计使 SceneNodeList 实际上是一个树状结构。在每个 SceneNode 中引用 CameraList、LightList、EntityList 中的元素实现了场景元素在 SceneNode 上的挂接。SceneManager 类还直接保存场景节点树的根节点指针以提供该树的访问入口。

重要函数

缺省构造函数

SceneManager();
缺省析构函数
virtual ~SceneManager();

摄像机相关函数
virtual Camera* createCamera(const String& name);
virtual Camera* getCamera(const String& name);
virtual void removeCamera(Camera *cam);
virtual void removeCamera(const String& name);
virtual void removeAllCameras(void);

光相关函数
virtual Light* createLight(const String& name);
virtual Light* getLight(const String& name);
virtual void removeLight(const String& name);
virtual void removeLight(Light* light);
virtual void removeAllLights(void);

材质相关函数
virtual Material* createMaterial(const String& name);
返回一个指向缺省材质设定的指针，用这个指针可以改变材质的设定
缺省设定如下：

- ambient = ColourValue::White
- diffuse = ColourValue::White
- specular = ColourValue::Black
- emmissive = ColourValue::Black
- shininess = 0
- No texture layers (& hence no textures)
- SourceBlendFactor = SBF_ONE
- DestBlendFactor = SBF_ZERO (no blend, replace with new colour)
- Depth buffer checking on
- Depth buffer writing on
- Depth buffer comparison function = CMPF_LESS_EQUAL
- Culling mode = CULL_CLOCKWISE
- Ambient lighting = ColourValue(0.5, 0.5, 0.5) (mid-grey)
- Dynamic lighting enabled
- Gourad shading mode
- Bilinear texture filtering

virtual Material* getDefaultMaterialSettings(void);
virtual void addMaterial(const Material& mat);
virtual Material* getMaterial(const String& name);
virtual Material* getMaterial(int handle);

场景节点相关函数
virtual SceneNode* createSceneNode(void);
virtual SceneNode* createSceneNode(const String& name);
virtual void destroySceneNode(const String& name);
取得根节点，在整个场景中只有一个根节点
virtual SceneNode* getRootSceneNode(void) const;
virtual SceneNode* getSceneNode(const String& name) const;

用 Mesh 创建一个实体
virtual Entity* createEntity(const String& entityName, const String& meshName);
用预先提供的 shape 创建实体
PrefabType 是一个枚举，提供预制的 shape，不需要模型
enum PrefabType {
    PT_PLANE
};
virtual Entity* createEntity(const String& entityName, PrefabType ptype);
virtual Entity* getEntity(const String& name);
销毁实体，注意实体必须没有被 SceneNode 所 attach，如果你不确定是否还有实体被 attach，使用 SceneManager::clearScene()
virtual void removeEntity(Entity* ent);
virtual void removeEntity(const String& name);
virtual void removeAllEntities(void);

清空场景，包括 SceneNodes，Cameras，Entities，Lights
virtual void clearScene(void);

设定环境光，缺省的环境光是 ColourValue::Black
void setAmbientLight(ColourValue colour);
ColourValue getAmbientLight(void);

设定世界坐标系
virtual void setWorldGeometry(const String& filename);

从 SceneManager 处取得一个推荐的 viewpoint，通常这个方法返回原点，除非通过 SceneManager::setWorldGeometry 设定了坐标系或者世界坐标系推荐了起始点。如果有多于一个的推荐点，将返回第一个，如果参数 random 为真，将随机的返回一个。
virtual ViewPoint getSuggestedViewpoint(bool random = false);

为 SceneManager 设定一个特殊实现的选项
virtual bool setOption(const String& strKey, const void* pValue) { return false; }
virtual bool getOption(const String& strKey, void* pDestValue) { return false; }
确认 SceneManager 是否有一个特殊实现的选项
virtual bool hasOption(const String& strKey) { return false; }
virtual bool getOptionValues(const String& strKey, std::list<SDDataChunk>& refValueList)

```
{ return false; }
virtual bool getOptionKeys( std::list<String>& refKeys ) { return false; }

Enables / Disables 一个天空面
virtual void setSkyPlane(
                bool enable,
                const Plane& plane,
const String& materialName,
Real scale = 1000,
                Real tiling = 10, bool drawFirst = true, Real bow = 0 );

Enables / Disables 一个天空盒
virtual void setSkyBox(
                bool enable,
const String& materialName,
Real distance = 5000,
                bool drawFirst = true,
const Quaternion& orientation = Quaternion::IDENTITY );

Enables / Disables 一个天空穹顶
        virtual void setSkyDome(
                bool enable,
const String& materialName,
Real curvature = 10,
                Real tiling = 8, Real distance = 4000, bool drawFirst = true,
                const Quaternion& orientation = Quaternion::IDENTITY);

雾相关函数
设置场景用的雾方式
        void setFog(
                FogMode mode = FOG_NONE,
ColourValue colour = ColourValue::White,
Real expDensity = 0.001,
Real linearStart = 0.0,
Real linearEnd = 1.0);

virtual FogMode getFogMode(void) const;
virtual const ColourValue& getFogColour(void) const;
virtual Real getFogStart(void) const;
virtual Real getFogEnd(void) const;
virtual Real getFogDensity(void) const;

公告板相关函数
virtual BillboardSet* createBillboardSet(const String& name, unsigned int poolSize = 20);
```

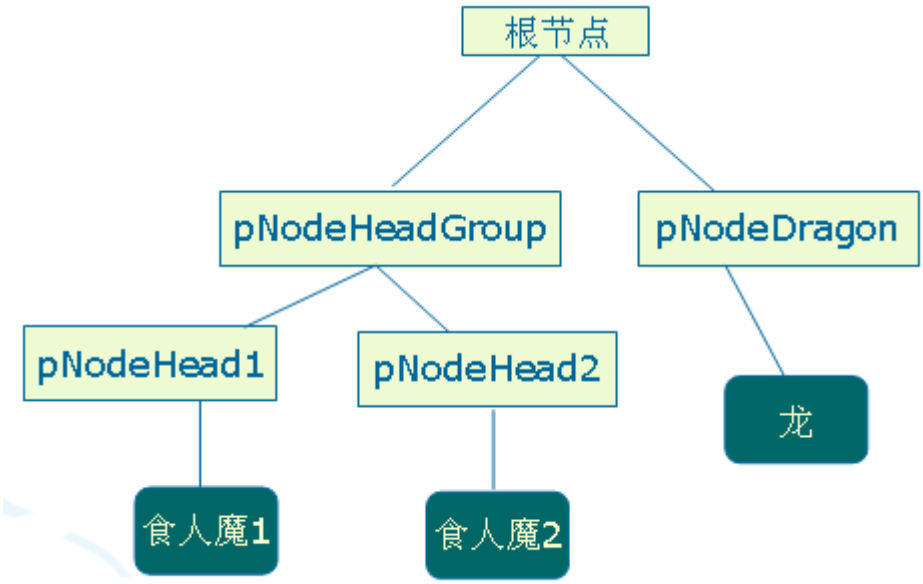
```
virtual BillboardSet* getBillboardSet(const String& name);
virtual void removeBillboardSet(BillboardSet* set);
virtual void removeBillboardSet(const String& name);

告知 SceneManager 是否渲染该节点
virtual void setDisplaySceneNodes(bool display);

动画相关函数
virtual Animation* createAnimation(const String& name, Real length);
virtual Animation* getAnimation(const String& name) const;
virtual void destroyAnimation(const String& name);
virtual void destroyAllAnimations(void);
virtual AnimationState* createAnimationState(const String& animName);
virtual AnimationState* getAnimationState(const String& animName);
virtual void destroyAnimationState(const String& name);
virtual void destroyAllAnimationStates(void);

手动渲染方法，高级用户使用
virtual void manualRender(
RenderOperation* rend, Material* mat, Viewport* vp,
                const Matrix4& worldMatrix, const Matrix4& viewMatrix, const Matrix4&
projMatrix) ;

Overlay 相关函数
virtual Overlay* createOverlay(const String& name, ushort zorder = 100);
virtual Overlay* getOverlay(const String& name);
virtual void destroyOverlay(const String& name);
virtual void destroyAllOverlays(void);

注册一个新的渲染队列监听器，当渲染队列被处理时将被更新
virtual void addRenderQueueListener(RenderQueueListener* newListener);
virtual void removeRenderQueueListener(RenderQueueListener* delListener);
```

场景管理器的使用举例

这个例子演示场景的组织和管理使用方法。程序运行时你会看到 2 个食人魔饶一条龙转动，同时它们自己也在自转。

思路

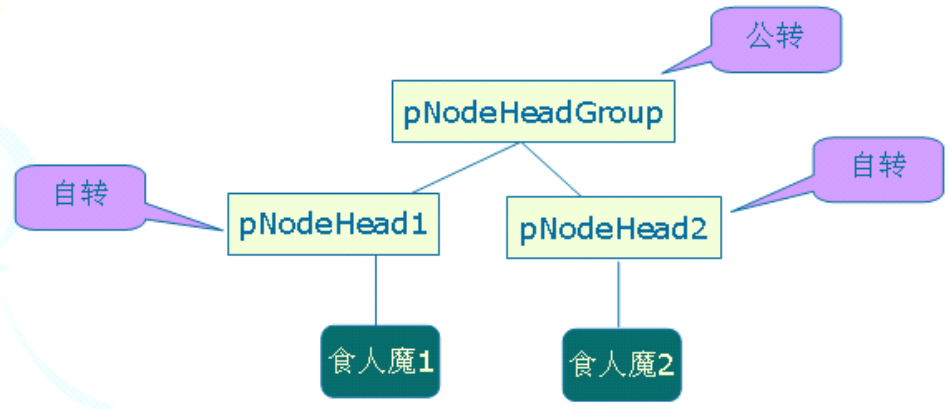
首先通过场景管理器在场景中建立如下的场景结构：



我们已经知道场景元素 Entity 仅仅是挂接到场景节点上，与位移、旋转和缩放有关的任务都由节点负责。那么龙和食人魔会出现在什么位置也就只与它们所在的节点有关。在 OGRE 系统中，根节点的缺省位置在场景中心（也就是 0，0，0 位置）。通过父场景节点建立子场景节点的时候，子节点相对于父节点的位移为 0，那么 pNodeDragon 和 pNodeHeadGroup 节点的位置也在场景中心，为了不让龙和 2 个食人魔挤在一起，我们把 pNodeHead1 和 pNodeHead2 节点分别相对于 pNodeHeadGroup 节点在 X 轴上位移正负 200 个单位长度。这样设置的结果将是龙出现在场景中心，而 2 个食人魔分别在龙的两边。

接下来是物体的运动，要使物体运动起来就必须在渲染过程中更改物体的位置和旋转量，这一点怎么办呢？不要忘了 OGRE 中有 FrameLisener 帧监听器的概念。FrameLisener 类中的 frameStarted 方法会在每一帧渲染之前被调用。OGRE 应用框架中的 ExampleFrameListener 已经实现了 frameStarted 方法，其主要工作是接受用户输入，控制摄象机。我们当然可以继承 ExampleFrameListener 类，扩充 frameStarted 方法让它控制物体的运动。

通过让 pNodeHeadGroup 每一帧都旋转一定的角度，可以带动其下的节点和节点挂接物体都旋转同样的角度，这样公转就实现了。用同样的原理，每一帧分别旋转 pNodeHead1 和 pNodeHead2 节点一定角度可以它们的自转。



部分代码

```
// Sample_SceneManager.h

// myFrameListener 类
class myFrameListener : public ExampleFrameListener
{
protected:
    // 对场景节点的引用，便于对运动的控制。
    SceneNode *pNodeHeadGroup,*pNodeHead1,*pNodeHead2;
public:
    myFrameListener(RenderWindow* win, Camera* cam, SceneNode* pNode1,SceneNode*
pNode2,SceneNode* pNode3)
        : ExampleFrameListener(win, cam)
    {
        pNodeHeadGroup = pNode1;
        pNodeHead1 = pNode2;
        pNodeHead2 = pNode3;
    }

    // 重新实现 frameStarted 函数，在这里实现对场景物体的运动控制
    bool frameStarted(const FrameEvent& evt)
    {

        // 旋转场景节点，当然也就旋转了它们下挂接的物体
        pNodeHeadGroup->yaw(evt.timeSinceLastFrame * 30);
        pNodeHead1->pitch(evt.timeSinceLastFrame * -360);
        pNodeHead2->roll(evt.timeSinceLastFrame * 1440);

        //  不要忘了，调用基类的 frameStarted 函数，以实现用户输入控制（摄象机漫游
控制）。
        return ExampleFrameListener::frameStarted(evt);

    }
};

// myApp 类
class myApp :public ExampleApplication
{
public:
    myApp(){}
protected:
```

```
SceneNode *pNodeDragon,*pNodeHeadGroup,*pNodeHead1,*pNodeHead2;
// 实现 createScene 函数，建立场景
void createScene(void)
{
    // 设置环境光
    mSceneMgr->setAmbientLight(ColourValue(1, 1, 1));

    // 在根节点下创建 pNodeDragon 节点
    pNodeDragon = mSceneMgr->getRootSceneNode()->createChild();
    // 创建“龙”Entity
    Entity *pEntityDragon = mSceneMgr->createEntity("Dragon", "Dragon.mesh");
    // 将龙 Entity 挂接到 pNodeDragon 节点下
    pNodeDragon->attachObject(pEntityDragon);
    // “龙”的模型太大了，缩小它所在的节点，该模型在显示时也就缩小了。
    pNodeDragon->setScale(0.5,0.5,0.5);

    // 创建 pNodeHeadGroup 场景节点
    pNodeHeadGroup = mSceneMgr->getRootSceneNode()->createChild();
    // 在 pNodeHeadGroup 场景节点下创建 pNodeHead1 节点
    pNodeHead1 = pNodeHeadGroup->createChild();
    // 使 pNodeHead1 节点偏离父节点
    pNodeHead1->translate(200,0,0);
    // 创建“食人魔”Entity。
    Entity *pEntityHead1 = mSceneMgr->createEntity("head1", "ogrehead.mesh");
    // 挂接该“食人魔”Entity 到 pNodeHead1
    pNodeHead1->attachObject(pEntityHead1);

    // 在 pNodeHeadGroup 节点下创建 pNodeHead2 节点
    pNodeHead2 = pNodeHeadGroup->createChild();
    // 使 pNodeHead2 节点偏离父节点
    pNodeHead2->translate(-200,0,0);
    // 由前面创造的“食人魔”Entity 克隆出另外一个“食人魔”Entity
    Entity *pEntityHead2 = pEntityHead1->clone("head2");
    // 将新的“食人魔”Entity 挂接到 pNodeHead2 节点下
    pNodeHead2->attachObject(pEntityHead2);

}

// 重新实现 createFrameListener 函数，因为基类 ExampleApplication 中的该函数创建的是 ExampleFrameListener 对象，而我们这里要使用 myFrameListener 对象。
void createFrameListener(void)
{
    mFrameListener= new myFrameListener(mWindow, mCamera, pNodeHeadGroup,pNodeHead1,pNodeHead2);
```

```
        mRoot->addFrameListener(mFrameListener);
    }

};
```

注：这里没有演示摄象机和光挂接到场景节点中的情况。这个示例中的的摄象机是在 ExampleApplication 类中定义的独立于场景节点树的摄象机。摄象机与 Entity 不一样，Entity 是完全不具有调整自身位置的能力，所以只能将自己挂接到场景节点上，通过对场景节点的位置调整来达到调整自身位置的目的，但摄象机有这个能力，所以可以独立控制。当然将摄象机挂接到场景节点上可以带来很多方便。如果要将摄象机和光挂接到场景节点上，方法与挂接 Entity 没什么区别，都是通过节点的 attachObject 函数来完成。attachObject 函数的参数是 MovableObject 类型的，而 MovableObject 类是 Entity、Camera 和 Light 类的基类，所以 attachObject 函数可以接收 Entity、Camera 和 Light 类型的参数。

摄象机

OGRE 中的摄象机支持透视投影（缺省投影方式、近大远小）和正射投影（大小与距离无关，是 CAD 设计中的常用投影方式）。摄象机还支持线画模式、纹理模式、灰度阴影模式等几种渲染模式。OGRE 场景中可以有多台摄象机，可以将摄象机“看到”的结果渲染到多个窗口，甚至还能实现分屏和画中画功能。OGRE 中的摄象机可以独立于场景节点树（摄象机本身也具有位置、旋转属性及控制方法），也可以被 attach 到场景节点上，通过对场景节点的控制来达到对摄象机的控制。

Camera 类

对摄象机的抽象。成员函数说明如下：

```
标准构造函数
Camera(String name, SceneManager* sm);
标准析构函数
virtual ~Camera();
返回渲染该摄像机的 scenemanager 的指针
SceneManager* getSceneManager(void) const;
取得摄像机的名字
virtual const String& getName(void) const;
设定投影模式（正射或透视），缺省为透视
void setProjectionType(ProjectionType pt);
取得使用的投影模式的信息
ProjectionType getProjectionType(void) const;
设定该摄像机需要的渲染细节级别
void setDetailLevel(SceneDetailLevel sd);
取得该摄像机的渲染细节级别
```

SceneDetailLevel getDetailLevel(void) const;
设定摄像机的位置
void setPosition(Real x, Real y, Real z);
void setPosition(const Vector3& vec);
取得摄像机的位置
const Vector3& getPosition(void) const;
移动摄像机
void move(const Vector3& vec);
void moveRelative(const Vector3& vec);
设定摄像机的方向向量
void setDirection(Real x, Real y, Real z);
void setDirection(const Vector3& vec);
取得摄像机的方向
Vector3 getDirection(void) const;
这是一个辅助方法用来自动计算摄像机的方向向量，在当前位置和所看的点，参数 targetPoint 是一个向量指明所看的点。
void lookAt(const Vector3& targetPoint);
void lookAt(Real x, Real y, Real z);
将摄像机绕 z 轴逆时针旋转指定角度
void roll(Real degrees);
绕 y 轴逆时针旋转指定角度
void yaw(Real degrees);
绕 x 轴上下逆时针旋转
void pitch(Real degrees);
旋转任意角度
void rotate(const Vector3& axis, Real degrees);
使用四元组绕任意轴旋转
void rotate(const Quaternion& q);
指定摄像机是绕本地 y 轴还是指定的固定轴旋转
void setFixedYawAxis(bool useFixed, const Vector3& fixedAxis = Vector3::UNIT_Y);
设定 y 方向的视野域，水平方向的视野域将依此计算
void setFOVy(Real fovy);
取得 y 方向的视野域
Real getFOVy(void) const;
设定到近裁减面的距离
void setNearClipDistance(Real nearDist);
取得到近裁减面的距离
Real getNearClipDistance(void) const;
设定到远裁减面的距离
void setAspectRatio(Real ratio);
取得当前纵横比
Real getAspectRatio(void) const;
内部使用，取得该摄像机的投影矩阵
const Matrix4& getProjectionMatrix(void);

```
内部使用，取得该摄像机的观察矩阵
const Matrix4& getViewMatrix(void);
取得平截台体的特定面
const Plane& getFrustumPlane( FrustumPlane plane );
测试给定的包容器是否在平截台体中
bool isVisible(const AxisAlignedBox& bound, FrustumPlane* culledBy = 0);
bool isVisible(const Sphere& bound, FrustumPlane* culledBy = 0);
测试给定的顶点是否在平截台体中
bool isVisible(const Vector3& vert, FrustumPlane* culledBy = 0);
返回摄像机的当前方向
const Quaternion& getOrientation(void) const;
设定摄像机的方向
void setOrientation(const Quaternion& q);
输出流功能
friend std::ostream& operator<<(std::ostream& o, Camera& c);
取得摄像机继承的方向，包括从附着节点继承的任何旋转
Quaternion getDerivedOrientation(void);
取得继承的位置，包括从附着节点继承的任何平移
Vector3 getDerivedPosition(void);
取得继承的方向向量
Vector3 getDerivedDirection(void);
覆盖 MovableObject 的方法
void _notifyCurrentCamera(Camera* cam);
const AxisAlignedBox& getBoundingBox(void) const;
void _updateRenderQueue(RenderQueue* queue);
const String getMovableType(void) const;


使能/使不能自动跟踪 scenenode
void setAutoTracking(bool enabled, SceneNode* target = 0,
                const Vector3& offset = Vector3::ZERO);
```

Camera 使用举例一

打开 OGRE 提供的 Demo_EnvMapping 那个例子程序，运行之。对于这个例子我们应该很熟悉了，通过键盘和鼠标可以控制摄象机在场景中漫游，那么摄象机的创建代码在哪里呢？从 EnvMapping.h 和 EnvMapping.cpp 中都找不到创建摄象机的代码！不要忘了我们是基于 OGRE 的应用框架建立的这个例子，在 OGRE 应用框架的 ExampleApplication.h 里为我们创建了摄象机，打开 ExampleApplication.h 文件可以发现如下函数：

```
virtual void createCamera(void)
{
```

```
    // 创建摄象机
    mCamera = mSceneMgr->createCamera("PlayerCam");
    // 将该摄象机放到 0,0,500 位置上
    mCamera->setPosition(Vector3(0,0,500));
    // 让摄象机“看”向 Z 轴负方向（从屏幕外向屏幕里）以模拟你的眼睛
    mCamera->lookAt(Vector3(0,0,-300));
    // 设置摄象机平截台体的“近面”距离
    mCamera->setNearClipDistance(5);

}
```

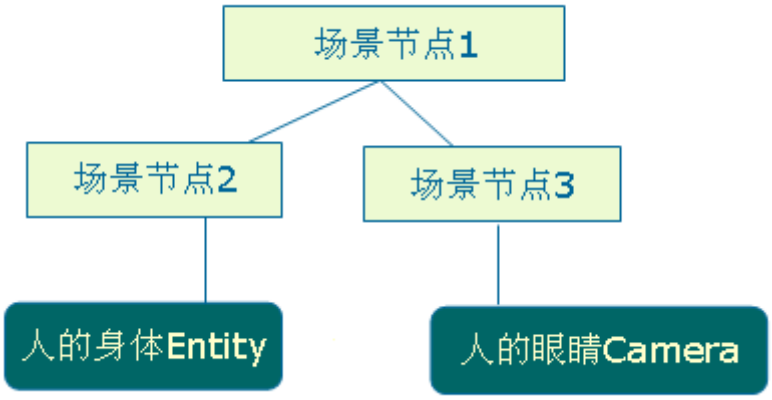
每一个通过 OGRE 应用框架创建的应用程序都会拥有一个通过 ExampleApplication 类的 createCamera 函数创建出来的摄象机，该摄象机站在 0,0,500 位置上看向场景中心。

摄象机的创建代码有了，那通过鼠标和键盘控制摄象机在场景中漫游的代码在哪里呢？在 OGRE 应用框架中 ExampleFrameListener 类的 frameStarted 函数里。该函数又调用 processUnbufferedInput 函数，我们可以在 processUnbufferedInput 函数中发现如下代码：

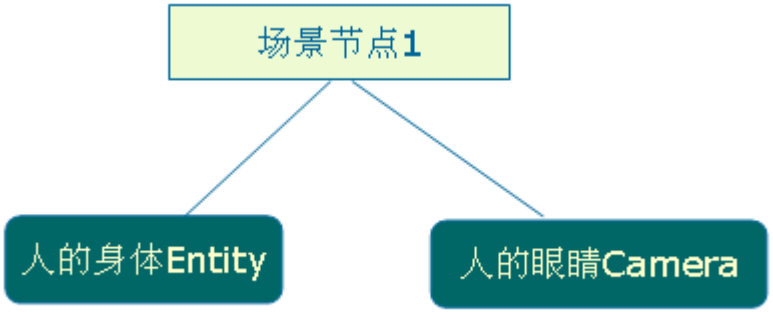
```
mInputDevice->capture();
……（省略若干行）
mCamera->yaw(rotX);
mCamera->pitch(rotY);
mCamera->moveRelative(vec);
```

首先获取鼠标状态，而后根据该状态计算摄象机的旋转和移动量，最后通过 Camera 的几个控制方法控制其运动。

注意到这里的摄象机是独立于场景节点树之外的。我们已经了解到场景节点树上可以挂接 Entity、摄象机和光。通过对场景节点的空间位置控制可以达到改变其下挂接的 Entity、摄象机和光的位置的目的。但注意 Entity 和摄象机不一样。在 OGRE 引擎的设计中 Entity 是完全没有移动、旋转等能力的，所以它只能把这些任务交给场景节点来完成，而摄象机具有移动和旋转函数，所以它并不一定要完全靠场景节点来完成这些任务。这就引出一个有趣的话题，摄象机放到场景节点中和不放进去的区别究竟有多大。一般来讲，摄象机如果不放在场景节点中，它就非常自由，程序员可以用程序任意控制它，就象在这个例子中一样。想象一下在 CS 中，你牺牲后，你依然可以控制你的眼睛（灵魂？摄象机？）在场景中穿墙过屋，并为同伴通风报信，就可以体会到这种自由。而如果把摄象机挂接到场景节点中，那么摄象机就和此节点和同在本节点下的其它 Entity 绑在一起了，一般在这种情况下就不再直接操作摄象机移动位置，而是和 Entity 一样交给场景节点来做。墙上来回转动的监视器就是由挂接在同一节点下的 Entity（监视器模型）和摄象机组成的。还有场景中的人，他们的身体(Entity)和眼睛(Camera)总是在一起，就因为他们同属于一个场景节点。



“人”的组织方法一



“人”的组织方法二

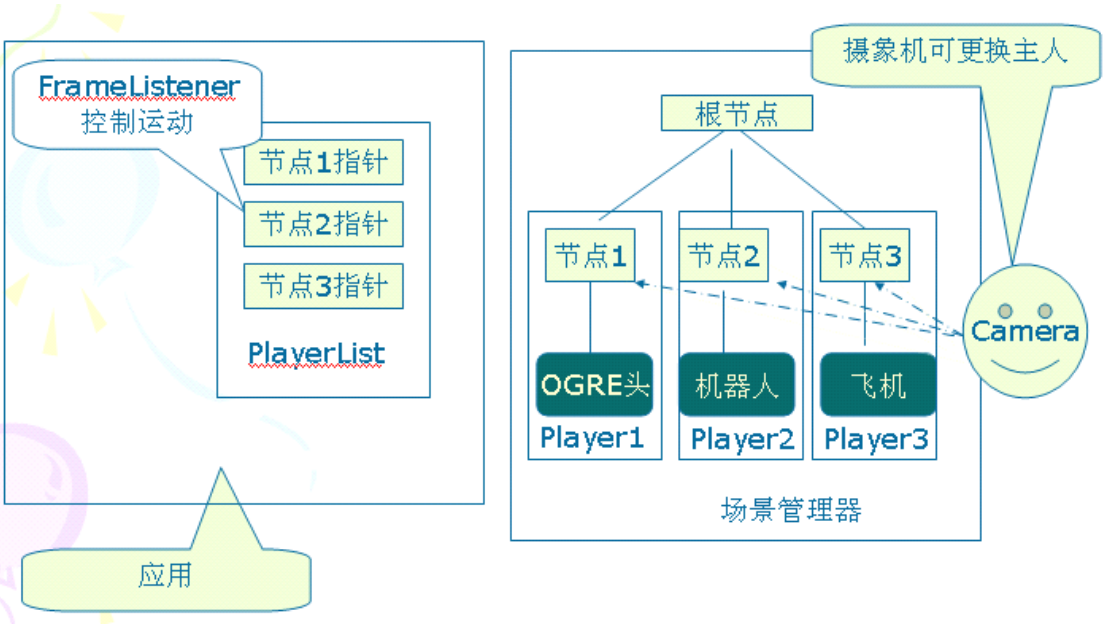
以上两图都将 Camera 放到了节点下，都可以实现身体和眼睛的同步。但第一种方法更好，因为第二种方法眼睛和身体同属于一个场景节点，它们之间无法实现相对位移，那么眼睛就可能会长在人的肚子里（节点的空间中心）。

将摄象机放到场景节点树中的做法使用也很普遍，下一个例子里我们将看到这样的情况。

Camera 使用举例二

思路

实现如下的场景节点树：



在该节点树中有一个食人魔、一个机器人和一架飞机。通过 FrameListener 来控制这三个 Player 都在自动旋转。通过按 TAB 键把 Camera 轮流挂接到三个 Player 所在的节点上，这样我们就会发现屏幕上会出现不同 Player 的以各自的视角所看到的世界。

为了便于对 Player 的控制，程序中使用一个 std::map 来保存 Player 列表，该列表中保存每个 Player 的所属节点名称和节点指针。

部分代码

```
// myExample.h

// 定义 PlayerList
typedef std::map<std::string,SceneNode*> PlayerList;

// 由应用框架中的 ExampleFrameListener 派生出 myFrameListener
class myFrameListener : public ExampleFrameListener
{
protected:
    // 接收 myapp 传过来的 Player 列表，以在这里控制其旋转
    PlayerList *mPlayerList;
    // 保存当前 Player 的迭代子
    PlayerList::iterator currentPlayer;
public:
    myFrameListener(RenderWindow* win, Camera* cam, PlayerList *pPlayerList)
        : ExampleFrameListener(win, cam)
    {
        mPlayerList = pPlayerList;
        // 缺省 Player 是列表中的第一人
        currentPlayer = mPlayerList->begin();
```

```
        // 将摄象机挂接到该 Player 所在的场景节点
        currentPlayer->second->attachCamera(mCamera);
    }

    bool frameStarted(const FrameEvent& evt)
    {
        // 对 TAB 键的反应
        if (mInputDevice->isKeyDown(KC_TAB))
        {
            // 把摄象机从当前 Player 上卸下来
            currentPlayer->second->detachObject(mCamera->getName());
            // 切换当前 Player
            currentPlayer++;
            if(currentPlayer == mPlayerList->end())
                currentPlayer = mPlayerList->begin();
            // 再把摄象机挂接到当前 Player 上来。
            currentPlayer->second->attachObject(mCamera);
        }

        // 让不同 Player 以不同的速度旋转
        mPlayerList->find("Robot")->second->yaw(evt.timeSinceLastFrame * 30);
        mPlayerList->find("Head")->second->yaw(evt.timeSinceLastFrame * -60);
        mPlayerList->find("Razor")->second->yaw(evt.timeSinceLastFrame * 120);

        // 调用基类的 frameStarted 函数
        return ExampleFrameListener::frameStarted(evt);

    }
};

// 由应用框架的 ExampleApplication 派生出 myApp
class myApp :public ExampleApplication
{
public:
    myApp(){}
protected:
    // Player 列表
    PlayerList mPlayerList;
    // 创建场景
    void createScene(void)
    {
        SceneNode *pNodeRobot,*pNodeHead,*pNodeRazor;
        // 设置环境光
        mSceneMgr->setAmbientLight(ColourValue(1, 1, 1));
```

```
        // 创建天空盒
        mSceneMgr->setSkyBox(true, "Examples/SpaceSkyBox", 50 );

        // 以下代码创建场景树
        // Create Robot Entity and attach it to a SceneNode
        pNodeRobot = mSceneMgr->getRootSceneNode()->createChild("Robot");
        Entity *pEntityRobot = mSceneMgr->createEntity("Robot", "Robot.mesh");
        pNodeRobot->attachObject(pEntityRobot);
        mPlayerList.insert(PlayerList::value_type(pNodeRobot->getName(),pNodeRobot));

        // Create OGREHead Entity and attach it to a SceneNode
        pNodeHead = mSceneMgr->getRootSceneNode()->createChild("Head");
        pNodeHead->translate(200,0,0);
        Entity *pEntityHead = mSceneMgr->createEntity("Head", "ogrehead.mesh");
        pNodeHead->attachObject(pEntityHead);
        mPlayerList.insert(PlayerList::value_type(pNodeHead->getName(),pNodeHead));


        // Create OGREHead Entity and attach it to a SceneNode
        pNodeRazor = mSceneMgr->getRootSceneNode()->createChild("Razor");
        pNodeRazor->translate(-200,0,0);
        // Create head1 entity and attach it to pNodeHead1
        Entity *pEntityRazor = mSceneMgr->createEntity("Razor", "Razor.mesh");
        pNodeRazor->attachObject(pEntityRazor);
        mPlayerList.insert(PlayerList::value_type(pNodeRazor->getName(),pNodeRazor));


    }

    //创建 myFrameListener
    void createFrameListener(void)
    {
        mFrameListener= new myFrameListener(mWindow, mCamera, &mPlayerList);
        mRoot->addFrameListener(mFrameListener);
    }
    // 重新实现基类的 createCamera 函数，关键是让摄象机与其所在场景节点的相对位置为
0，100，0。即高 100 个长度单位，防止摄象机在 Entity 的肚子里出现。
    virtual void createCamera(void)
    {
        // Create the camera
        mCamera = mSceneMgr->createCamera("PlayerCam");

        // 设置摄象机位置
        //mCamera->setPosition(Vector3(0,0,500));
```

```
        mCamera->setPosition(Vector3(0,100,0));
        // Look back along -Z
        mCamera->lookAt(Vector3(0,0,-300));
        mCamera->setNearClipDistance(5);

    }

};
```

为了让例子简单一点，这里采用的是前面讲的第二种眼睛与身体的组合方法，摄象机与 Entity 的相对位置是靠摄象机的 setPosition 函数完成的，这样做并不是一个很好的方法。建议大家将本例改为前面讲的第一种眼睛与身体的组合方法，将摄象机与 Entity 的相对位置关系交给场景节点去做，那样摄象机的位置就可以设置为 0,0,0。

事情还没有结束，因为摄象机是属于 Player 的了，我们就不能让键盘再控制摄象机将他移出身体以外，所以需要更改 ExampleFrameListener.h 中 frameStarted 函数的代码，因为 ExampleFrameListener.h 是 OGRE 应用框架的一部分，所以请注意 copy 该文件，再更改。

找到 frameStarted 函数中的如下代码：

```
        mCamera->yaw(rotX);
        mCamera->pitch(rotY);
        mCamera->moveRelative(vec);
```

将最后一行注释掉，即可以让摄象机可以受鼠标控制旋转（东张西望？），但不能移动。

光

材质与材质脚本

基本概念

Ogre 的材质（Material）

为了优化渲染，必须把渲染状态的变化减少到最小。而最频繁的渲染状态改变是材料的变化（大多是纹理的变化）。

Ogre 的 Material 类封装了物体的所有材料属性，类似于 3D Studio 中 material 的概念。平时不被认为是属于材料的属性，像 culling 模式和深度缓存设置等，也被 Material 包含近来了。因为这些属性同样影响了物体的外观，把它们放到 Material 类里可以集中设置所有影响物体的属性。这和 D3D 中只保存颜色组件而没有纹理映射的 Material 有明显不同。Ogre 的 Material 可以被认为是'Shader'的等同物。

Material 类包括如下几类属性：

1、 基本的表面材质属性，如对不同颜色的反射率、Shininess 等等：

2、 组成 Material 的纹理层

3、 纹理之间的混合（Blend）方式
4、 深度缓冲设置
5、 Culling 模式
6、 纹理过滤方式（三线性过滤、双线性过滤）
7、 是否受光照影响
8、 Shading 选项
9、 雾化

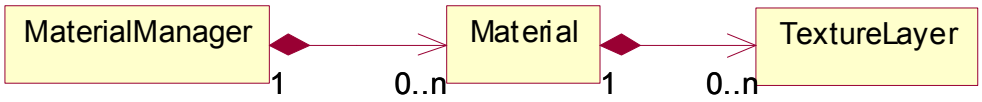
其中第二条的纹理层可以有多个，Ogre 在 Material 类的内部定义了 TextureLayer 类。

纹理层（TextureLayer）

一个纹理层可以是一幅静止的图像，也可以是一幅以某种方式运动的图像，还可以是由多幅图像组成的动画。它可以实现多种纹理特效，如 BUMPMAP、环境帖图、运动的纹理。

材质管理器（MaterialManager）

MaterialManager 类负责管理 Material 库。和材质相关的类图如下：



MaterialManager 还负责分析 Material 脚本（Material Script），从而初始化 Material 的属性。下面我们来分析一下 Ogre 提供的材质脚本语言。

Ogre 的材质脚本

Ogre 提供的材质脚本可以帮助开发者简单的设置又酷又眩的材质特效，而不用重新编译。当然你也可以用 Material 和 TextureLayer 类提供的诸多成员函数来做到，但这就有点不太实用了。

Ogre 材质脚本的默认扩展名为.material，你也可以通过 MaterialManager 类的 parseAllSources 方法来规定新的扩展名。系统初始化时会自动分析所有的材质脚本文件，并设置材质的属性。注意，这里只设置材质的属性，并不会将材质的纹理调入内存，因为那样会招致极大的内存消耗！

示例一：环境帖图

在场景中显示一个 OGRE 头，并设置其表面材质为环境帖图，该环境帖图的属性在 Example.material 中设置。我们先看一下 createScene()函数的代码：

```
void createScene(void)
{
    //设置环境光
    mSceneMgr->setAmbientLight(ColourValue(0.5, 0.5, 0.5));
```

```
    //创建一个实体（OGRE 头）
    Entity *ent = mSceneMgr->createEntity("head", "ogrehead.mesh");

    //设置实体的材质
    ent->setMaterialName("Examples/EnvMappedRustySteel");

    // 将实体附属到场景根结点上
    mSceneMgr->getRootSceneNode()->createChild()->attachObject(ent);
}
```

在这里，Entity 的成员函数 setMaterialName 是关键，字符串类型的参数对应材质脚本中的材质名。"Examples/EnvMappedRustySteel"就是材质名，它在 Example.material 文件中定义了环境帖图，脚本代码如下：

```
//材质名
Examples/EnvMappedRustySteel
{
    //环境光
    ambient 1.0, 1.0, 1.0
    //散射光
    diffuse 1.0, 1.0, 1.0
    shading phong
    //纹理层0
    {
        //纹理图象文件
        texture RustySteel.jpg
    }
    //纹理层1
    {
        //纹理图象文件
        texture spheremap.png
        //纹理层之间的颜色累加
        colour_op add
        //指定环境帖图的方式
        env_map spherical
    }
}
```

从这个例子中我们可以发现使用脚本来定义材质非常方便。首先，脚本采用类 C++的语法，以｛｝作为分隔符，以//作为注释符。每一个 Material 必须有一个名字，在这个例子中就是 Examples/EnvMappedRustySteel，它对应一个材质。脚本规定一个命令占一行，不可以串行。

Ogre 为其材质脚本定义了许多关键字(命令),用来设置材质属性。如上例中的“ambient”与“diffuse”就是关健字，分别设置材质对环境光和散射光的反射率。在 Material 定义中嵌套一个“{}”，就代表一个纹理层，你可以在大括号中定义纹理层的属性，上例中有两层纹理，下层简单的显示了一幅名为“RustySteel.jpg”的图像 ，上层纹理采用了环境帖图并设

定了两层纹理的混合方式为颜色相加。

材质脚本的关健字有很多，具体请参考 Ogre Tutorials 的“Material Script”一节。

示例二：Example.material

分析并尝试更改 Ogre 运行环境中的 Example.material 文件，并在 Ogre 提供的演示 TextureFx.exe 中观察效果。下面列出的是 TextureFx.exe 中用到的三个材质的定义：

```
// 流动的水的特效
Examples/TextureEffect3
{
    ambient 0.2 0.2 0.2

    // 关闭硬件拣选与软件拣选，即双面渲染
    // 硬件拣选，由硬件渲染器负责。基于画三角形的方式（顺时针，逆时针）。
<clockwise|anitclockwise|none>
    // 软件拣选，在前几个信息改善到硬件器之前，进行拣选。基于三角形的前和后（由三角形的向量决定）。
    cull_hardware none
    cull_software none

    // 水用了两层纹理
    // 纹理层0
    {
        texture Water01.jpg
        scroll_anim -0.25 0.1
    }
    // 纹理层1
    {
        texture Water01.jpg
        scroll_anim -0.1 0.25
        // 纹理层1与纹理层2之间的混合方式，add代表两层颜色相加。
        colour_op add
    }
}

// 大小正弦波影响的特效
Examples/TextureEffect1
{
    ambient 0.75 0.75 0.75

    // 关闭硬件拣选与软件拣选，即双面渲染
    // 硬件拣选，由硬件渲染器负责。基于画三角形的方式（顺时针，逆时针）。
    // 软件拣选，在前几何信息改善到硬件器之前，进行拣选。基于三角形的前和后（由三角形的向量决定）。
```

```
    cull_hardware none
    cull_software none

    // 纹理层0
    {
        // 纹理图的文件名
        texture BumpyMetal.jpg

        // wave_xform指定一个波形函数，来对指定的纹理层属性（scale或scroll或rotate）
产生影响
        // wave_xform <纹理层的某个属性> <波形> <base> <频率> <相位> <振幅>
        // 下面这句可以解释成：纹理的宽度值  =  纹理的宽度值  *  正弦波当前时间的
值(base+sine)
        wave_xform scale_x sine 0 0.1 0 5

        // 旋转速率  =  旋转速率  +  正弦波当前的值
        // wave_xform rotate sine 0.1 0.1 0 5
    }
}


// 纹理流动的管道
Examples/TextureEffect2
{
    scene_blend add
    {
        texture Water02.jpg
        scroll_anim 0.5 0
    }
}
```

材质脚本关健字说明

ambient 设置材质的环境光反射属性

格式: ambient <red> <green> <blue>
正确的参数在 0.0 和 1.0 之间取值。直接影响物体材质对环境光反射能力。默认值为白色（1.0 1.0 1.0）。

diffuse 设置材质的漫反射属性

格式: diffuse <red> <green> <blue>
正确的参数在 0.0 和 1.0 之间取值。直接影响物体材质对漫射光的反射属性。默认值为白色（1.0 1.0 1.0）。

specular 设置材质的镜面反射属性

格式: specular <red> <green> <blue> <shininess>
正确的颜色参数在 0.0 和 1.0 之间取值，shininess 属性可以是任何正数。直接影响物体材质的镜面反射属性。默认值为无镜面反射（0.0 0.0 0.0 0.0）。

emissive 设置材质本身的发光程度

格式: emissive <red> <green> <blue>
正确的颜色参数在 0.0 和 1.0 之间取值。 如果一个物体自发光，它将不需要外界的照明，但是，值得注意的是这不表明这个物体将会成为一个光源：它只会照亮自己。默认是黑色（0.0 0.0 0.0）。

scene_blend 设置与场景的混合方式，有两种形式

格式 1: scene_blend <add|modulate|alpha_blend>
这个格式比较简单常用一些，参数意义如下：

add	渲染的结果将被以相加的方式加入场景之中，与 scene_blend one one 等价。对爆炸，火焰，光照，幽灵等效果比较好。
Modulate	渲染的结果与场景相乘。对烟、玻璃杯和单个的透明物效果较好。与 scene_blend src_colour one_minus_src_colour 等价。
alpha_blend	渲染结果中的 Alpha 成员将被用作遮罩。与 scene_blend src_alpha one_minus_src_alpha 等价。

格式 2: scene_blend <src_factor> <dest_factor>
这个格式比较麻烦，但是比较完善。结果的计算公式为(texture * sourceFactor) + (scene_pixel * destFactor)其中 sourceFactor 和 destFactor 如下：

One	常数 1.0
Zero	常数 0.0
dest_colour	当前点的颜色
src_colour	纹理对应点的颜色
one_minus_dest_colour	1 - (dest_colour)
one_minus_src_colour	1 - (src_colour)
dest_alpha	当前点的 Alpha 值
src_alpha	纹理对应点的 Alpha 值
one_minus_dest_alpha	1 - (dest_alpha)
one_minus_src_alpha	1 - (src_alpha)

默认值: scene_blend one zero (不透明)

depth_check 是否开深度测试

格式: depth_check <on|off>
默认打开深度缓存。有助于判断两个点的遮挡关系和前后关系，体现三维立体感。

depth_write 是否允许对已经存在的深度缓存进行写操作

格式: depth_write <on|off>

默认允许，关掉的话，则被关的 Material 会一直浮动在所有物体前面。

depth_func 当深度缓存打开的时候，挑选一个比较函数

格式: depth_func <func>

always_fail	从不比较
always_pass	总是用新的换掉旧的
less	新的比旧的小就换掉
Less_equal	新的小于等于旧的就换掉
equal	等于就换掉
not_equal	不等于就换掉
greater_equal	新的大于等于旧的就换掉
greater	新的比旧的大就换掉

默认为:小于等于就换掉 depth_func less_equal

cull_hardware 设置硬件 Cull 模式

格式: cull_hardware <clockwise|anitclockwise|none>

默认顺时针 Cull。这与 OpenGL 的默认是一样的，但和 D3D 的默认相反。（因为 Ogre 用的是 OpenGL 采用的右手坐标系）

cull_software 设置软件 Cull 模式

格式: cull_software <back|front|none>

默认背面。相当于硬件 Cull 模式的顺时针。

lighting 光照

设置动态光照是否为此材质打开。如果关掉，将使材质本身的所有的 ambient, diffuse, specular, emissive 和 shading 属性无效，仅仅与外界的光照有关。

格式: lighting <on|off>

默认: lighting on

shading 阴影模式

格式: shading <flat|gouraud|phong>

Flat	每个表面仅仅用一个颜色填充
gouraud	线性过渡表面颜色
phong	并非所有的硬件都支持，这种模式测定每一个顶点的颜色。

默认: shading gouraud

filtering 设置纹理过滤方式

格式: filtering <none|bilinear|trilinear>

默认是双线性（bilinear）

Texture Layer 专用属性

texture 设置纹理要使用的图名

格式: texture <texturename>

无默认值，必须指定一个纹理名。

anim_texture 动画纹理

设置动画纹理使用的图片文件名。

格式 1 (短的): anim_texture <base_name> <num_frames> <duration>

指定一个图片名称，以这个名称后缀_1、_2 一直到_num（由 num_frames 指定），duration 指定间隔时间。

格式 2 (长的): anim_texture <frame1> <frame2> ... <duration>

一个一个指定图片名称，duration 指定间隔时间。

无默认值

粒子系统及粒子脚本

粒子系统在三维显示中占有很重要的地位，如 3D 中的雨，雪，喷泉，爆炸效果等都是粒子系统神奇魅力的表现。

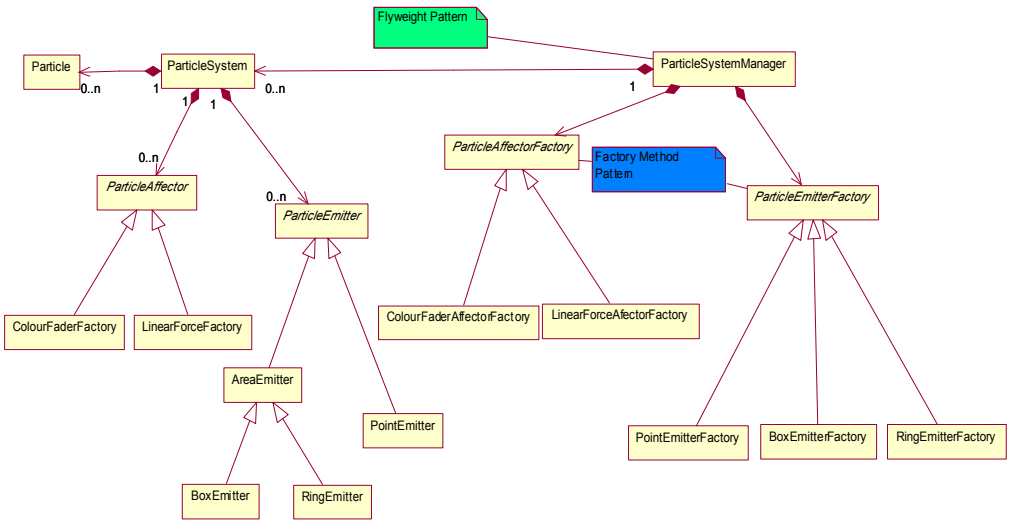
我们可以用程序的方式虚拟美国哥伦比亚号航天飞机在太空中爆炸的场景：大大小小的碎片向四处飞溅，耀眼的火花，滚滚的浓烟，在这其中多数元素都是混乱的。从技术的角度讲本场景中的大多数效果得益于一个优秀的粒子系统。烟，火花，这些 3D 中的效果通常使用粒子系统来创建的。

基本概念

粒子是用四边形来表示的。它有长宽，和其方向，有数量和材质，还有重量。这些属性封在粒子属性变换器（Affector）中，Affector 中包含颜色衰退变换器（ ColourFaderAffector）和线形影响变换器（LinearForceAffector）。粒子的产生由粒子发生器（Emmiter）产生。它包含有盒状粒子发生器（BoxEmmitter）和点状粒子发生器（ PointEmmiter），由这些组成粒子脚本文件，以.particle 为其扩展名。点状粒子发生器随机地从单一的点发射粒子；盒状粒子发生器随机的从一个区域发射粒子。通过创建 Plugins 你可以向 ogre 里增加一个新的发射器。当前 ogre 只支持点状粒子发生器和盒状粒子发射器。

当粒子发生器不停的喷发出大量粒子时，就可以形成烟、火和爆炸等效果。

粒子系统结构图：



粒子系统脚本

OGRE 提供了粒子系统脚本语言，可以在脚本中设置粒子的各种属性，而不用重新编译程序。这样给使用者带来了方便。需要注意的粒子系统脚本文件有自己的语法，类似 C++：注释用“//”，用“{ }”来区分界限。但不支持在代码后跟注释。

粒子系统在脚本中是以模板的形式定义的，你可以在程序运行期创建多个实例。

载入粒子脚本

粒子系统脚本是在系统初始化的时候载入的，缺省情况下系统在公共资源路径（Root::addResourceLocation 函数指定）下查找所有扩展名为“.particle”的文件并解析它们。如果你想指定其它扩展名可使用 ParticleSystemManager::getSingleton().parseAllSources 方法，如果想解析单个的粒子系统脚本文件可使用 ParticleSystemManager::getSingleton().parseScript 方法。

一旦粒子系统脚本被载入，你可以使用 ParticleSystemManager::getSingleton().createSystem()方法来创建一个实例化的粒子系统，该方法接收两个参数，一个该粒子系统的名称，而另外一个参数是要参照的模板名称（也就是脚本中定义过的模板名称。

格式

可以在一个脚本文件中定义多个粒子系统模板。

以下是一个典型的粒子脚本，该脚本包含三个粒子系统模板，它被用于 OGRE 的粒子 DEMO 中，你可以运行该 DEMO 来查看实际效果。当然你也可以修改这些属性来改变效果。

```
// 粒子系统模板名
// Exudes greeny particles which float upwards
Examples/GreenyNimbus
{
    // 粒子的 Material
    material            Examples/Flare
    // 粒子的宽度
    particle_width   30
    // 粒子的高度
    particle_height 30
    // 粒子的裁剪模式：整体包围盒裁剪或单个粒子裁剪。
    cull_each           false
    // 粒子数目
    quota               10000
    // 公告板的类型：point 指代表粒子的四边形总是面向摄像机。
    billboard_type   point

// 盒状粒子发射器
    emitter Box
    {
            // 粒子发射时偏离 direction 的最大角度
        angle               30
            // 发射速率(个/秒)
        emission_rate    30
            // 粒子生存时间(秒)
        time_to_live      50
```

```
            // 粒子的发射方向
        direction           0 1 0
            // 速率
        velocity          10
            // 颜色起始值
        colour_range_start    1 1 0
            // 颜色结束值(在起始值和结束值之间取随机数)
        colour_range_end       0.3 1 0.3
            // 定义粒子发射器 BOX 的大小
        width                  60
        height                 60
        depth                  60
    }

    // LinearForce：对运动中的粒子的加上一个外力，影响其运动轨迹。
    // Make em float upwards
    affector LinearForce
    {
            // 指定外力的影响(用向量表示)
        force_vector          0 100 0
            // add：粒子的运动向量加上外力的向量。效果：匀加速运动。
        force_application add
    }

    // ColourFader：影响粒子中的颜色
    // Fader
    affector ColourFader
    {
            // 每秒衰减 0.25
        red -0.25
        green -0.25
        blue -0.25
    }

}

// A sparkly purple fountain
Examples/PurpleFountain
{
    material            Examples/Flare2
    particle_width    20
    particle_height 20
    cull_each           false
    quota                10000
```

```
    billboard_type    oriented_self

    // Area emitter
    emitter Point
    {
        angle                15
        emission_rate    75
        time_to_live     3
        direction            0 1 0
        velocity_min     250
        velocity_max     300
       //颜色变换上下限
        colour_range_start    0 0 0
        colour_range_end      1 1 1
    }

    // Gravity
    affector LinearForce
    {
        force_vector          0 -100 0
        force_application add
    }

    // Fader
    affector ColourFader
    {
        red -0.25
        green -0.25
        blue -0.25
    }
}


// A downpour
Examples/Rain
{
    material             Examples/Droplet
    particle_width   50
    particle_height 100
    cull_each            true
    quota                10000
    // Make common direction straight down (faster than self oriented)
    billboard_type    oriented_common
    common_direction 0 -1 0
```

```
        // Area emitter
        emitter Box
        {
            angle                   0
            emission_rate       100
            time_to_live        5
            direction           0 -1 0
            velocity            50
            colour_range_start  0.3 1 0.3
            colour_range_end        0.7 1 0.7
            width                   1000
            height                  1000
            depth                   0
        }

        // Gravity
        affector LinearForce
        {
            force_vector            0 -200 0
            force_application add
        }

    }
```

脚本中的每一个粒子系统模板都必须有一个名字，且必须是在“{”前的第一行。这个名称必须是唯一的。名字中可以包含“/”来构成路径，但 OGRE 引擎只把它当成字符串看待，并不真正来分析这个路径，它仅仅方便程序员来区分层次。

一个粒子系统可以设置一些上层属性，如：quota 表示允许的最大粒子个数。除了基本属性外，还必须在一个粒子系统模板内嵌套定义发射器 Emitters 和属性变换器 affectors。它们内部的属性与它们的类型有关。

粒子系统脚本的属性较多，请参考 OGRE Tutorial 中的 Particle Scripts 部分。

示例一

打开 OGRE DEMO 中 Emample.particle 试着改变粒子发射器（emitter Box 和 emitter Point）和粒子的发射速率（比如将 emission_rate　100 改为 emission_rate　30），粒子生存时间(秒)（time_to_live　）等等，注意区分大小写，存盘，然后再运行 Demo_ParticleFX.exe 看看效果。

示例二

打开 OGRE 中的 Demo_ParticleFX 工程，查看其中的代码，理解粒子系统的调用方法。

该程序在运行时，场景中有一个食人魔，头上冒着绿色的烟雾（粒子系统），旁边有两个运动的粒子发生器喷发着五颜六色光点（还是粒子系统）。

OGRE 引擎在初始化的时候会自动载入粒子系统脚本文件，我们不用管它。

在程序的 createScene 函数里，通过 ParticleSystem* pSys1 = ParticleSystemManager::getSingleton().createSystem("Nimbus", "Examples/GreenyNimbus"); 创建出一个实际的粒子系统 Nimbus，它采用的模板"Examples/GreenyNimbus"就是在粒子系统脚本中定义的。

本程序共有三个粒子系统，创建方法都一样，只不过采用的模板不同。

粒子系统被创建好后，可以将其 attach 到场景节点上去，便于运动控制。代码如下：

mSceneMgr->getRootSceneNode()->createChild()->attachObject(pSys1);

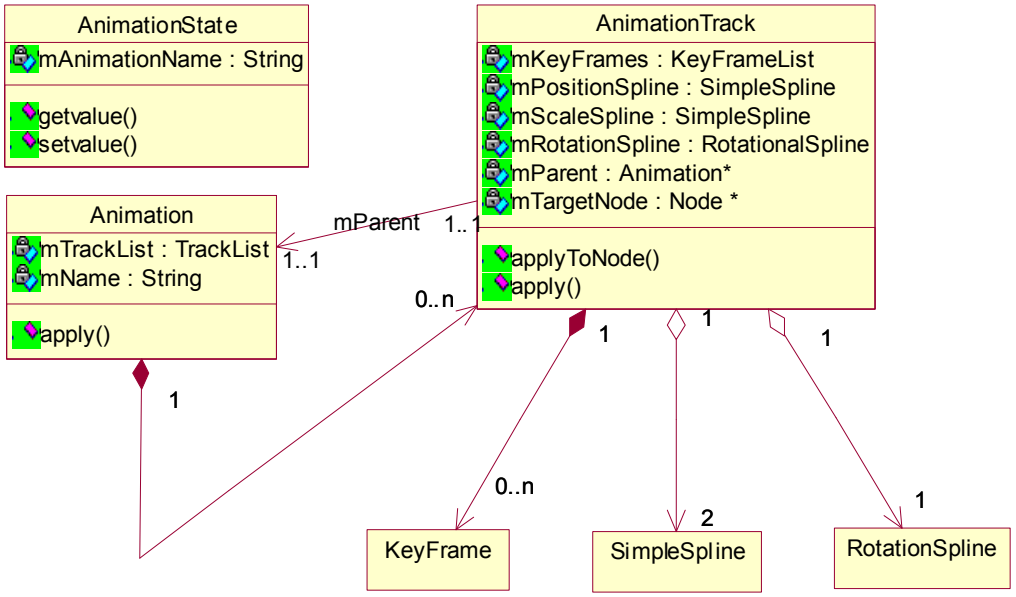
一旦粒子系统被 attach 到场景节点上，就可以通过 FrameListener 的 frameStarted 方法来控制节点运动，当然粒子也就边运动边喷发了。

动画基础

实现动画的基本原理：多个关键帧组成一个动画轨迹，多个动画轨迹组成一个动画。在各个关键帧之间通过插值算法（Spline 插值或线性插值）进行插值来生成最终的动画。一个动画不只包括关键帧，还有其它的一些属性(动画名、动画长度、动画的权重、是否被启用等)。一个动画轨迹可以指定其控制的节点。

OGRE 中与基本动画相关的类

根据动画原理，我们抽象出以下几个类：



SimpleSpline 与 RotationSpline 类

实现了样条的插值算法。SimpleSpline 用来对 Translate 和 Scale 进行插值，RotationSpline 用来对 Rotation 进行插值。

关键帧类（KeyFrame）

组成动画最基本的元素。一个动画轨迹里有多个关键帧，每个关键帧具有自己的位置、缩放比例和旋转角，同时每个关键帧还保存有自己在整个动画轨迹里所处的时间点。在实际运行时，根据当前时间，通过对两个关键帧的插值可以得到当前帧（当前位置、缩放比例和旋转角）。随着时间的变化，插值得到的当前帧也是变化的，动画就产生了。由于关键帧包括位置信息、缩放比例信息和旋转信息，所以可以实现运动动画、缩放动画和旋转动画以及混合动画。

动画轨迹类（AnimationTrack）

动画轨迹由多个关健帧组成，负责对相邻的两个关键帧之间插值。

每个动画轨迹保存关键帧列表、本动画轨迹所属的动画（Animation）、本动画轨迹所控制的 Node，又由于动画轨迹负责关键帧的插值，所以它保存 SimpleSpline（简单样条插值计算器）和 RotationSpline（旋转样条插值计算器）对象以实现插值功能。

一个动画轨迹可以对一个物体产生作用，使物体按照动画轨迹运动。这个物体可以是一个 Node（一块骨头或场景中的一个结点）。如果骨头受动画轨迹控制，可以实现骨骼动画，如果场景节点受动画轨迹控制，可以直接实现场景节点的动画运动。

重要函数

创建一个关键帧，传入这一帧的所在的时间点。

KeyFrame* createKeyFrame(Real timePos);

根据当前时间，得到插值计算出来的的当前帧。

KeyFrame getInterpolatedKeyFrame(Real timeIndex) const;

使当前动画轨迹对其控制的节点产生作用，参数是当前时间点、权重和是否累计权重。

void apply(Real timePos, Real weight = 1.0, bool accumulate = false);

动画类（Animation）

一个动画由多个动画轨迹（AnimationTrack）组成。而一个动画轨迹可以控制一个节点，这样一个动画可以使多个节点沿着自己的轨迹运动。

设想一下一个人的行走动画，人身上的关节点都沿着自己的轨迹运动，如果把两个关节点连接起来，就形成骨头，再让表面的网格受骨头影响而运动，骨骼动画的基础有了！

每个动画保存自己的 AnimationTrack 列表，保存动画名称和长度（时间）。

重要函数

设置关键帧间的插值方式。参数 IM_LINEAR 代表线性插值、IM_SPLINE 代表样条插值。

void setInterpolationMode(InterpolationMode im);

创建一个动画轨迹。第一个参数是这个动画轨迹的唯一标识，第二个参数指定应用这个动画轨迹的节点。

AnimationTrack* createTrack(unsigned short handle, Node* node);

使当前动画对其控制的节点产生作用（委托 AnimationTrack 进行），参数是当前时间点、权重和是否累计权重。

void apply(Real timePos, Real weight = 1.0, bool accumulate = false);

动画状态类（AnimationState）

一个动画状态类的对象对应一个动画类的对象，看上面的类图，AnimationState 的数据成员 mAnimationName 与 Animation 类的数据成员 mName 是相对应的。它保存相应动画的状态。

动画状态包括动画名、当前时间点，动画长度（总时间）、动画权重和动画的 enable 开关。

重要函数

设置动画是否启用。

void setEnabled(bool enabled);

移动当前时间点，让动画状态在动画时间线上向前移动。参数为移动量。

void addTime(Real offset);

场景管理器类（SceneManager）

场景管理器负责动画和动画状态的创建与维护。在场景管理器里保存有mAnimationsList（动画列表）和 mAnimationStates（动画状态列表），其中每个动画和动画状态通过名字一一对应。

场景管理器中有一个很重要的方法_applySceneAnimations ，它在每次渲染时（_renderScene 函数里）都会被调用（自动调用不需要程序员控制），它的任务是在每一帧渲染前根据动画状态更新动画，完成动作。_applySceneAnimations 方法遍历全部的动画状态，并根据这些状态找出与之一一对应的动画，再找出每个动画中的全部动画轨迹，在每个轨迹里都保存有该轨迹控制的节点。_applySceneAnimations 方法先将这些被动画控制的节点都还原为初始状态，而后再调用动画的 apply 方法将全部节点更新到新的位置、大小或方向，从而使动画向前进行。

通过 FrameLisener 调用动画状态的 addTime 方法，可以完成动画状态的更新。场景管理器的_applySceneAnimations 方法会根据新的动画状态将对应动画的相关节点的位置、大小和方向也更新，这就是 OGRE 中实现动画的基本原理和方法。

重要函数

创建动画 Animation，参数为动画名和长度（时间）

virtual Animation* createAnimation(const String& name, Real length);

创建动画状态 AnimationState，参数是与动画对应的名称。

virtual AnimationState* createAnimationState(const String& animName);

基本动画实例：

定义一个 10 秒种的动画，这个动画包含一个动画轨迹（上下翻转）。让这个动画应用到当前摄像机上去，程序运行时，我（第一人称摄像机）应该在上下翻转。

在 createScene 函数里做初始化工作。

首先我们考虑怎样可以把动画应用到当前摄像机上。因为一个动画可以应用到一个节点上，所以可以创建一个节点并将当前摄像机 attach 到这个节点上去，代码如下：

```
SceneNode* camNode = mSceneMgr->getRootSceneNode()->createChild();
camNode->attachObject(mCamera);
```

下面定义动画、动画轨迹以及关键帧：

```
// 定义动画，指定动画的名称及长度（这里为 10 秒）
Animation* anim = mSceneMgr->createAnimation("CameraTrack", 10);
```

```
// 指定动画关键帧之间的插值方式（包括线性插值和样条插值）
anim->setInterpolationMode(Animation::IM_SPLINE);
// 定义动画的一个动画轨迹，并指定这个轨迹是作用到 camNode 节点上的
AnimationTrack* track = anim->createTrack(0, camNode);
// 定义动画轨迹包含的关键帧，下面定义了四个关键帧，加上起始帧
// 五个关健帧形成了一个翻转的动画。
KeyFrame* key = track->createKeyFrame(0); // startposition
key = track->createKeyFrame(2.5);
key->setTranslate(Vector3(500,500,-1000));
key = track->createKeyFrame(5);
key->setTranslate(Vector3(-1500,1000,-600));
key = track->createKeyFrame(7.5);
key->setTranslate(Vector3(0,-100,0));
key = track->createKeyFrame(10);
key->setTranslate(Vector3(0,0,0));
```

然后定义 AnimationState 类的对象，它和刚才定义的动画类相对应。设置动画的状态为启用：

```
mAnimState = mSceneMgr->createAnimationState("CameraTrack");
mAnimState->setEnabled(true);    // 启用该动画
```

到此，初始化工作就做完了。

最后，要想使动画动起来，我们需要重载 ExampleFrameLisener 类的 frameStarted 函数，并调用下面的函数，根据传入的时间来设置动画的状态：mAnimState->addTime(evt.timeSinceLastFrame);

骨骼动画

基本概念

什么是骨骼动画

骨骼动画用骨架（由一系列骨头构成的继承体系）来单独保存动作信息，这样就将动作信息与网络、皮肤信息分割成两种数据结构分别进行处理，从而比以往的动画技术效率更高。每块骨头都有自己的位置和旋转方向，这些骨头以树的形式组织起来构成骨骼。例如：腕关节是肘关节的子节点，肘关节又是肩关节的子节点。肩关节旋转会自动带动肘关节运动，腕关节同样也会运动。

那么怎么使一个网格产生动作效果呢？我们可以使网格上的每个顶点都对应于一块或多块骨头，当这些骨头移动时便会影响网格的位置。如果一个点与多块骨关联，则必须通过指定权重来决定每块骨头对此顶点的影响程度（一个顶点对应一块骨头时，该点的权重为 1.0 ）。

与关键帧动画相比，使用骨骼动画有很多优点。首先，骨骼动画需要保存的动作数据量非常小。其次，通过指定不同的权重，可以很容易的将多个动作绑定在一起形成新的动作。还可以实现动作间的平滑过渡等等。

当然骨骼动画的实现中，骨骼本身的运动还是需要关键帧来完成，但信息量已经很小了。

Ogre 的骨骼动画

Ogre 骨骼信息和动画信息保存到后缀名为.skeleton 的文件中，你可以通过 OGRE 提供的导出插件工具（Milkshape 和 3Dmax 的 exporter）来导出该类型的文件。当你创建基于.Mesh 文件的 Entity 时，.Skeleton 文件将会自动被系统加载进来。为了操作方便，Entity 自动给每一个动作指定一个 AnimationState 类（请参考基本动画）的对象，你可以通过 Entity::getAnimationState 函数来得到具体的动作。

示例一：行走的机器人

该实例创建基于 robot.mesh 文件的实体（Entity）对象，指定其“走”动作（动作信息都在.skeleton 文件里），并将其显示到屏幕上。robot.mesh 文件中保存机器人的网格信息，Entity 会自动加载保存有机器人骨骼信息的 robot.skeleton 文件。我们重载 ExampleApplication 类的 createScene 函数，其部分代码如下：

```
void createScene(void)
{
    // ....
    // 设置关键帧之间的插值方法为样条插值
    Animation::setDefaultInterpolationMode(Animation::IM_SPLINE);
    // 创建基于网格文件 robot.mesh 的 Entity
    Entity *ent = mSceneMgr->createEntity("robot", "robot.mesh");
    // 将实体附属到场景根结点上
    mSceneMgr->getRootSceneNode()->createChild()->attachObject(ent);
    // 得到实体“走”动作的 AnimationState 类对象
    mAnimState = ent->getAnimationState("Walk");
    // “Enable”（起始）该动作
```

```
        mAnimState->setEnabled(true);
    }
```

在 createScene 函数里成功的指定了机器人的当前动作为“走”，下一步要让动画“动”起来，还需要根据时间跨度计算当前帧的骨骼位置。重载 ExampleFrameListener 类的 frameStarted 函数：

```
    bool frameStarted(const FrameEvent& evt)
    {
        // 将两帧之间的时间差传入 AnimationState::addTime 函数，该函数内部会计算出动
        // 画的当前时间点。
        mAnimState->addTime(evt.timeSinceLastFrame);

        // 调用父类的 frameStarted 函数
        return ExampleFrameListener::frameStarted(evt);
    }
```

示例二：控制机器人的动作

界面

这里的界面是指菜单、HUD 及提示信息框在内的综合体。

基本概念

Overlay

将被渲染在“普通”场景内容之上的层。Overlay 是那些将在主场景被渲染之后才渲染的可视组件的容器。这些可视组件将构成 HUD（heads-up-display）、菜单或其它在主场景内容之上的任何东西。

一个 Overlay 总是占满一个 viewport 的全部尺寸，尽管它包含的组件并没有那么大，那么多。Overlay 并不代表任何可视组件，它只是一个可视组件的容器。

Overlay 可以通过调用 SceneManager::createOverlay 函数来创建，或者把它们定义在一个脚本文件里(.overlay files)，引擎会自动解析这些脚本，并创建它们。你可以定义无限多个 Overlay。一个 overlay 被创建之后是不可见的，你必须调用 show 函数让它们显示出来。这样你就可以预先定义很多 Overlay（比如菜单），在需要它们的时候才将它们显示出来。在同一时刻可以存在和显示多个 Overlay，Overlay 的 zorder 属性决定了谁在上谁在下。

缺省情况下，Overlay 会被渲染到全部的 viewport。当你只有一个全屏 viewport 的时候这非常好用，但是当你在程序中用几个 viewport 构成“画中画”的时候，你可能不想在小画面中也显示 Overlay。你可以通过调用 viewport 的 Viewport::setDisplayOverlays 方法将某些 viewport 的 Overlay 显示状态关闭。

Overlay 支持旋转、卷动和缩放。通过 Overlay::scroll, Overlay::rotate and Overlay::scale 这几个函数可以达到目的。

Overlay 中可以包括 2D 元素，也可以包括 3D 元素。

GuiElement

将在 Overlay 中被显示的 2D 元素。这个类抽象了在 overlay 中出现的 2D 元素的全部信息。事实上，并非所有的 GuiElement 的实例都可以被加入到 Overlay 中，只有 GuiContainer 类（GuiElement 类的派生类）的实例才可以。GuiContainer 对象可以包含 GuiElement 对象。也就是说一般情况下，Overlay 包含一个或多个 GuiContainer，而 GuiContainer 包含 GuiElement。

GuiElements 被 GuiManager 管理。GuiManager 负责创建和删除 GuiElements，并且负责从 plugins 接收新类型的 GuiElements。

GuiElements 属性中的位置和尺寸表达方式（标尺模式）有两种：Pixel Mode 模式和 Relative Mode 模式。当采用 Pixel Mode 模式时，位置和尺寸用 Pixel 为单位。用这种方式屏幕元素的位置和大小将与屏幕分辨率有关。例如：800*600 分辨率下，某个 GuiElements 的左上角坐标为 10，10，大小为 50，50。那么它将显示在屏幕左上方。但当分辨率变为 1024*768 的时候，它的位置将向左上移动一点，且看起来会变小。当采用 Relative Mode 模式时，GuiElements 中的位置和尺寸是用 0.0 - 1.0 的参数表达的，是一个和屏幕宽度及高度的比值。这样做的方式是为了使这些值与显示分辨率无关，从而使显示结果可以适应任何分辨率。例如：0.5x0.5 的尺寸代表屏幕宽和高的一半大，此外请注意，0.5x0.5 的尺寸在屏幕上显示出来并不是一个正方形，因为屏幕本身就不是正方形。

GuiElements 被设计成可扩展的，它是 StringInterface 类的子类，所以它的参数也是通用参数。

我们可以继承 GuiElements 类，实现自己的功能复杂 GuiElements。一般情况下通过写 plugin 来扩展 GuiElements。

GuiContainer

可以包含其它 GuiElement 实例的特殊 GuiElement。GuiContainer 类其实是 GuiElement 类的派生类。它也是可以被直接 attach 到 Overlay 上的最小元素。GuiContainers 也由 GuiManager 管理。GuiManager 负责创建和删除 GuiElements，并且负责从 plugins 接收新类型的 GuiElements。

向 Overlay 加入一个 GuiContainer 的方法很简单，调用 Overlay 的 add2D 函数就可以了。

向一个 GuiContainer 加入子元素的方法也很简单，调用 GuiContainer 的 addChild 函数就可以了。子元素可以是 GuiContainer，也可以是 GuiElements。通过一级一级加入子元素可以构成一个树状结构。需要注意的是，子元素的位置是相对于父元素的。

GuiElementFactory

GuiElementFactory：创建 GuiElement 的工厂类。该类是一个抽象类，任何一个具体 GuiElement 的工厂都必须继承它，并实现其中的接口。

GuiManager

它的任务是管理 GuiElement（及其子类）的实例的生命周期，并从 plugin 模块中注册新的 GuiElement 类型。

TextAreaGuiElement

2D 界面元素，GuiElement 的直接派生类，定义界面中的文本区域。

PanelGuiElement

2D 界面元素，GuiContainer 的直接派生类，定义界面中的面板区域，在面板中还能包括其它界面元素。

BorderPanelGuiElement

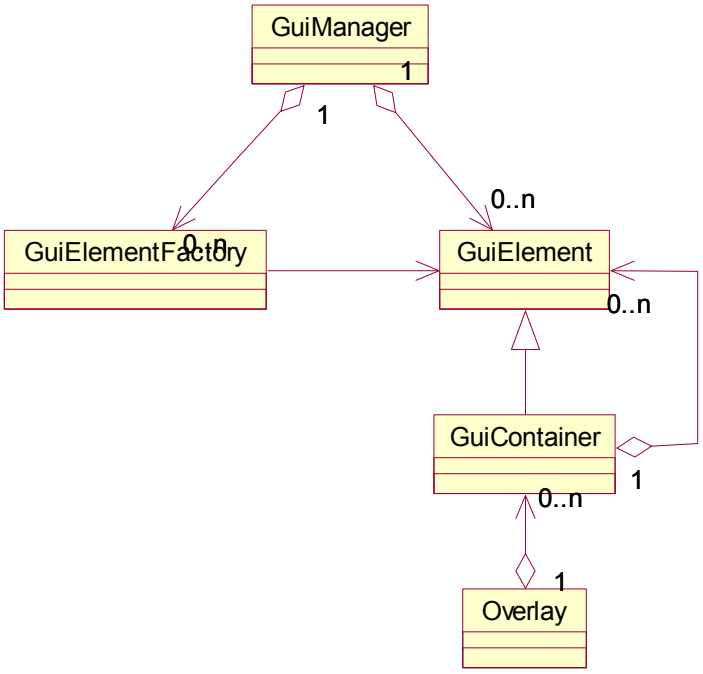
2D 界面元素，GuiContainer 的直接派生类，定义界面中的带边面板区域，在带边面板中还能包括其它界面元素。

ButtonGuiElement

2D 界面元素，BorderPanelGuiElement 的直接派生类，定义界面中的按钮。

ListGuiElement

2D 界面元素，PanelGuiElement 的直接派生类，定义界面中的列表选择框。



GUI 静态结构图

Overlay 脚本

Overlay 脚本为我们提供了使用脚本来定义可重用的 overlays 的手段。

载入脚本

Overlay 脚本在系统初始化的时候被载入。缺省情况下，系统会把公共资源路径（参考 Root::addResourceLocation）下所有扩展名为'.overlay'的文件都载入并解析。如果你的 Overlay 脚本使用了其它扩展名，你可以使用 OverlayManager::getSingleton().parseAllSources 函数载入它们，如果你想载入单个 Overlay 脚本，就使用 OverlayManager::getSingleton().parseScript 函数。

格式

可以在一个脚本文件中定义多个 overlay。脚本采用伪 C++格式，用{ }分块，注释采用‘//’（注意不支持注释嵌套），而且支持模板继承。以下是一个典型的实例：

```
// The name of the overlay comes first
MyOverlays/ANewOverlay
{
   zorder 200

   container Panel(MyGuiElements/TestPanel)
```

```
    {
        // Center it horzontally, put it at the top
        left 0.25
        top 0
        width 0.5
        height 0.1
        material MyMaterials/APanelMaterial

        // Another panel nested in this one
        container Panel(MyGuiElements/AnotherPanel)
        {
            left 0
            top 0
            width 0.1
            height 0.1
            material MyMaterials/NestedPanel
        }
    }

}
```

以上的例子定义了一个名为“MyOverlays/ANewOverlay”的 overlay，其中包括 2 个嵌套的 panels。它使用缺省的相对标尺模式（用 0-1 的数来表示长度和位置，0-1 的数代表与屏幕宽度和高度的比值）。

脚本中的每一个 overlay 都必须有名字，且必须在“{”之前一行指定。名字不能重复。名字中可以包含“路径”格式（用/分割），用以区分层次和避免重名，但 OGRE 引擎并不把这些名字解析成一个层次结构，它仅仅是一个字符串。

在大括号中间是 overlay 的属性和其他嵌套的元素。本例中，overlay 只有一个'zorder'属性，'zorder'用于为重叠的 overlay 区分覆盖关系，zorder 值大的将渲染在上面。

在 overlay 中加入元素

在 overlay 里可以包括 2D 和 3D 元素。这些元素必须以如下关键字开头：

'element' 不能再嵌套其它元素的 2D 元素。

'container' 可嵌套 2D 元素和 container 的容器。

'entity' 3D 元素，只能放在 overlay 里，不能嵌套在其它容器中。

通过'container'可以实现 2D 元素的层层嵌套。

'container' 和 'element' 块

以如下格式开头：

[container | element] <type_name> (<instance_name>) [: <template_name>]

{ ...

type_name:：GuiElement 的类型名，必须在 GuiManager 中注册。OGRE 引擎的 Plugin_GuiElements.dll 中已提供的类型有：Panel、 BorderPanel 、TextArea、Button 和 List。你可以自己写 Plugin 扩充其它类型。

instance_name ： 标识本元素的唯一的名称。可以通过 GuiManager::getSingleton().getGuiElement(name)获取到指定名称的元素指针。

template_name:：可选参数，指定模板名称。

块中的属性取决于每个元素类型，以下属性是每个元素都有的：

metrics_mode	标尺模式
horz_align	横向对齐
vert_align	纵向对齐
left	左边位置
top	上边位置
width	宽度
height	高度
material	材质
caption	标题

TextArea 特有的属性：

font_name	字体名
char_height	字符高度
colour_top	上部颜色
colour_bottom	下部颜色

BorderPanel 特有的属性：

border_size
border_material
border_topleft_uv
border_top_uv
border_topright_uv
border_left_uv
border_right_uv
border_bottomleft_uv
border_bottom_uv
border_bottomright_uv

Button特有的属性：

border_up_material	按钮弹起状态材质
border_down_material	按钮按下状态材质

List 特有的属性：

item_template	选项文字模板
v_spacing	选项间的纵向距离
h_spacing	
item_material_selected	选项被选中后的材质

'entity' 块

必须以以下格式开头：

entity <mesh_name> (<entity_name>)

{ ...

mesh_name: .mesh 文件名

entity_name: 唯一的 entity 名

你还可以定义如下属性

position

rotation

注意：材质名可以从.mesh 文件中读出，当然你可以指定为.material 中定义的其它材质。

Templates

你可以使用模板来定义大量具有相同属性的元素。模板是不能加入到 overlay 中的抽象元素。各种元素可以继承它并获得它定义的缺省属性。在元素（container, element, or entity）的定义前加'template'关键字就可以定义模板。模板元素一般定义在脚本文件的开头，而且不能定义在 Overlay 里面。建议把模板元素定义在独立的脚本文件中，这样可以很容易实现重用和个性化调整。

元素可以象 C++那样用“：”符号继承自模板。“：”放在元素名称的右括号后。模板名称又放在“：”符号后。

A template can contain template children which are created when the template is subclassed and instantiated. Using the template keyword for the children of a template is optional but recommended for clarity, as the children of a template are always going to be templates themselves.

```
template container BorderPanel(MyTemplates/BasicBorderPanel)
{
  left 0
  top 0
  width 1
  height 1
    // setup the texture UVs for a borderpanel
    // do this in a template so it doesn't need to be redone everywhere
  material Core/StatsBlockCenter
  border_size 0.05 0.05 0.06665 0.06665
  border_material Core/StatsBlockBorder
  border_topleft_uv 0.0000 1.0000 0.1914 0.7969
  border_top_uv 0.1914 1.0000 0.8086 0.7969
  border_topright_uv 0.8086 1.0000 1.0000 0.7969
  border_left_uv 0.0000 0.7969 0.1914 0.2148
```

```
    border_right_uv 0.8086 0.7969 1.0000 0.2148
    border_bottomleft_uv 0.0000 0.2148 0.1914 0.0000
    border_bottom_uv 0.1914 0.2148 0.8086 0.0000
    border_bottomright_uv 0.8086 0.2148 1.0000 0.0000
}
template container Button(MyTemplates/BasicButton) : MyTemplates/BasicBorderPanel
{
    left 0.82
    top 0.45
    width 0.16
    height 0.13
    material Core/StatsBlockCenter
    border_up_material Core/StatsBlockBorder/Up
    border_down_material Core/StatsBlockBorder/Down
}
template element TextArea(MyTemplates/BasicText)
{
    font_name Ogre
    char_height 0.08
    colour_top 1 1 0
    colour_bottom 1 0.2 0.2
    left 0.03
    top 0.02
    width 0.12
    height 0.09
}
        MyOverlays/AnotherOverlay
{
    zorder 490
    container BorderPanel(MyElements/BackPanel) : MyTemplates/BasicBorderPanel
    {
        left 0
        top 0
        width 1
        height 1

        container Button(MyElements/HostButton) : MyTemplates/BasicButton
        {
            left 0.82
            top 0.45
            caption MyTemplates/BasicText HOST
        }

        container Button(MyElements/JoinButton) : MyTemplates/BasicButton
```

```
        {
            left 0.82
            top 0.60
            caption MyTemplates/BasicText JOIN
        }
    }
}
```

以上的例子使用模板来创建一个按钮。注意到 button 模板继承自 borderPanel 模板，这减少了创建按钮时需要设置的属性。

还注意到 Button 的实例用模板名来设置 caption 属性。模板也可以被 elements 用于动态创建子 elements（按钮创建了一个 TextAreaElement 来作为自己的 caption）。

Overlay 实例一

查看 OGRE 运行环境中的 Ogre.overlay 文件，在 OGRE 自带的 DEMO 中都使用了这个文件中定义的 overlay。

FPS 的数据是如何显示到 overlay 中的？？？？

Overlay 实例二

在 Ogre.overlay 的基础上，加两个新的 overlay。其中包括一个 OGRE 的 LOGO。另一个 overlay 包括一个 3D 元素。程序运行时包括 LOGO 的 overlay 来回运动。

思路

overlay 中即可以包含 2D 元素也可以包含 3D 元素。通过 overlay 的 setScroll 函数可以其运动。

部分代码

Ogre.overlay 文件中加入如下脚本，注意备份原文件。

```
// A silly example of how you would do a 3D cockpit
Examples/KnotCockpit
{
    zorder 100
    entity knot.mesh(hudKnot)
    {
        position 0 0 -50
        rotation 0 0 0 0
    }
```

```
}
// another logo overlay
Examples/MyLogo
{
    zorder 100
    container Panel(Examples/LogoPanel)
    {
        metrics_mode pixels
        horz_align right
        vert_align top
        top 20
        left -165
        width 160
        height 85
        material Core/OgreText
    }

}
```

这部分脚本定义了两个 Overlay。一个带 3D 元素一个带 2D 元素。

以下是 myapp.h 文件

```
#include "ExampleApplication.h"
class myFrameListener : public ExampleFrameListener
{
protected:
    Overlay * pMyOverlayLogo;
public:
    myFrameListener(RenderWindow* win, Camera* cam)
        : ExampleFrameListener(win, cam)
    {
        pMyOverlayLogo = (Overlay*)
OverlayManager::getSingleton().getByName("Examples/MyLogo");
        pMyOverlayLogo->show();
    }
    // 重新实现 frameStarted 函数，在这里控制 Overlay 的运动
    bool frameStarted(const FrameEvent& evt)
    {
        static float angle = 0.0;
        if( angle >= Math::TWO_PI)
            angle = 0.0;
        angle += 0.1;
        // 在 X 轴按正弦函数方式运动
        pMyOverlayLogo->setScroll(Math::Sin(angle),0);
        //  不要忘了，调用基类的 frameStarted 函数，以实现用户输入控制（摄象机
```

```
漫游控制）。
            return ExampleFrameListener::frameStarted(evt);
        }
};

class EnvMapApplication : public ExampleApplication
{
public:
    EnvMapApplication() {}
protected:

    // Just override the mandatory create scene method
    void createScene(void)
    {
        // Set ambient light
        mSceneMgr->setAmbientLight(ColourValue(0.5, 0.5, 0.5));
        // Create a point light
        Light* l = mSceneMgr->createLight("MainLight");
        // Accept default settings: point light, white diffuse, just set position
        // NB I could attach the light to a SceneNode if I wanted it to move automatically
with
        //    other objects, but I don't
        l->setPosition(20,80,50);
        Entity *ent = mSceneMgr->createEntity("head", "ogrehead.mesh");
        // Set material loaded from Example.material
        ent->setMaterialName("Examples/EnvMappedRustySteel");
        // Add entity to the root scene node
        mSceneMgr->getRootSceneNode()->createChild()->attachObject(ent);
        // 获取指定名称的 Overlay 指针
        Overlay * pMyOverlayLogo = (Overlay*)
OverlayManager::getSingleton().getByName("Examples/MyLogo");
        // 缺省情况下 Overlay 被载入后是不显示的，让其显示。
        pMyOverlayLogo->show();
        // 获取指定名称的 Overlay 指针
        Overlay * pMyOverlayKnot = (Overlay*)
OverlayManager::getSingleton().getByName("Examples/KnotCockpit");
        // 缺省情况下 Overlay 被载入后是不显示的，让其显示。
        pMyOverlayKnot->show();

    }

    void createFrameListener(void)
    {
        mFrameListener= new myFrameListener(mWindow, mCamera);
```

```
        mRoot->addFrameListener(mFrameListener);
    }

};
```

OGRE 会在程序运行时自动载入.overlay 文件，缺省情况下是不显示的。在 OGRE 应用框架的 ExampleApplication 类中把 DebugOverlay 的显示开关打开了，其它的 overlay 必须由程序员手工打开。以上程序在 createScene 函数中通过 OverlayManager::getSingleton().getByName（）函数获取到 overlay 的指针，再通过 show 函数打开显示开关。

Overlay 提供滚动、旋转和缩放函数，在 FrameListener 控制其滚动很好地达到了动画效果。

实现界面事件处理

OGRE 引擎已经将一些界面元素的基本事件处理完成，例如：按钮被点击时的动作效果、List 列表中选项被选择后的特殊显示等。但是按钮被按下后还要完成什么事情需要程序员自己完成，例如 Exit 按钮被按下后程序要退出、Option 按钮被按下后要进入系统设置菜单等。

OGRE 的事件处理目前还未全部完成，请暂时参考 OGRE 的 Demo_Gui。

复杂场景

OGRE 除了我们已经见过的普通场景以外还支持大型复杂的场景，如：宽阔的野外地形和复杂的大楼和迷宫。3D 引擎支持大型复杂场景的主要难度在于对场景的组织和裁减。一个大型场景的三角型数量极其庞大，如果没有有效的场景组织和裁减方法，计算机在渲染的时候效率会很低。解决大办法就是用 BSP 或八叉树等数据结构将场景组织起来，利用与这些数据结构相关的快速检索算法将摄象机看到的内容拣选出来，再送到渲染器进行渲染，这样图形处理器的负载才会减小。

在 OGRE 中场景管理器的类型有四种：

```
ST_GENERIC,             普通场景
ST_EXTERIOR_CLOSE,      室外封闭场景
ST_EXTERIOR_FAR,        室外无限场景
ST_INTERIOR             室内场景
```

除了第一种普通场景外，其它几种都是处理复杂场景的专用场景管理器。OGRE 引擎中有一个 SceneManagerEnumerator 类，它的作用是管理已实现的场景管理器。在 OGRE 的 Root 类里就就聚合了一个 SceneManagerEnumerator 对象，已供选择需要的场景管理器。在 OGRE FrameWork 的 ExampleApplication 类中可以看到这样的代码：

```
virtual void chooseSceneManager(void)
{
    // Get the SceneManager, in this case a generic one
    mSceneMgr = mRoot->getSceneManager(ST_GENERIC);
```

}

从以上代码可知，在普通情况下使用的是普通场景管理器。如果要使用特殊的场景管理器，通过 mRoot->getSceneManager（）函数做出选择就可以了。

在 OGRE 的 SceneManager 类中有这样的一个函数 setWorldGeometry，它的作用是读入世界信息（复杂场景中的大楼、野外地形等不变场景内容，区别于程序员手工加入的可以控制的场景元素），并将其管理起来。对于普通场景管理器来讲，没有“World Surface(Terrain,Room...)”这个概念，调用这个函数将只会抛出一个"World geometry is not supported by the generic SceneManager. 的异常。而 OGRE 在其引擎提供的 Plugin_BSPSceneManager.dll 中提供了通过 BSP 算法实现的 ST_INTERIOR 室内场景管理器，它重新实现了 setWorldGeometry 函数，使其可以读入一个.bsp 的室内场景文件并管理之。.bsp 是 QUAKE 和 CS 等室内游戏使用的场景文件类型。OGRE 还在其引擎提供的 Plugin_OctreeSceneManager.dll 中提供了通过八叉树算法实现的 ST_EXTERIOR_CLOSE 室外封闭场景管理器，它同样重新实现了 setWorldGeometry 函数，使其可以读入一个.cfg 室外场景配置文件，并载入以灰度图形式表达的地形高程图。

需要注意的是特殊场景管理器不仅可以管理复杂场景中那些属于“World Surface”（复杂场景中的大楼、野外地形等不变场景内容）的场景内容，它也具有普通场景管理器的全部功能。所以在应用时，通过 setWorldGeometry 函数载入“不变世界”之后依然可以向场景中加入 SceneNode 和 Entity 等其它场景元素，依然可以通过 FrameLisener 来控制这些普通场景元素的运动。只不过它们将在一个具有楼房或山坡的场景中运动了。

室内场景

Demo_BSP 工程是 OGRE 引擎带的室内场景例子。打开 Demo_BSP 工程查看代码。

OGRE 在其引擎提供的 Plugin_BSPSceneManager.dll 插件中提供了通过 BSP 算法实现的 ST_INTERIOR 室内场景管理器。OGRE 引擎在初始化的时候会载入 Plugins.cfg 中指定的全部插件，只要在 Plugins.cfg 中包含 Plugin_BSPSceneManager.dll 插件就可以使用了。

由于 OGRE 目前支持的是 QUAKE3 的地图，所以必须有一个 quake3settings.cfg 来指定 QUAKE3 的地图包（.pk3 文件，其实是一个 ZIP 文件）和其中的地图文件名（.bsp）。QUAKE3 是一个商业游戏软件，其中的全部地图的知识产权都属于 ID 公司，所以 OGRE 引擎的 DEMO 程序并没有附带任何 QUAKE3 地图，你必须在自己的计算机上安装 QUAKE3 的地图包，并将路径设置到 quake3settings.cfg 中去，Demo_BSP 才能够正确运行。

在 BSP.h 文件中，定义了 BspApplication 类，该类的构造函数首先读入 quake3settings.cfg，并解析出其中的地图包文件名和地图文件名。代码如下：

```
BspApplication()
{
    // Load Quake3 locations from a file
    ConfigFile cf;
    cf.load("quake3settings.cfg");
    mQuakePk3 = cf.getSetting("Pak0Location");
    mQuakeLevel = cf.getSetting("Map");
}
```

地图包文件实际上是一个 ZIP 文件，该 ZIP 文件中包含 OGRE 真正需要的实际扩展名为.bsp 的地图文件，所以还必须将地图包 ZIP 文件加入到 OGRE 的资源搜索路径中去。代

码如下：

```
 void setupResources(void)
{
    ExampleApplication::setupResources();
    ResourceManager::addCommonArchiveEx(mQuakePk3, "Zip");


}
```

由于采用了特殊的室内场景管理器，所以必须重新实现 chooseSceneManager 函数，以选择正确的场景管理器。代码如下：

```
void chooseSceneManager(void)
{
    mSceneMgr = mRoot->getSceneManager(ST_INTERIOR);
}
```

最后是创建场景，代码如下：

```
void createScene(void)
{
    // Load world geometry
    mSceneMgr->setWorldGeometry(mQuakeLevel);

    // modify camera for close work
    mCamera->setNearClipDistance(4);
    mCamera->setFarClipDistance(4000);

    // Also change position, and set Quake-type orientation
    // Get random player start point
    ViewPoint vp = mSceneMgr->getSuggestedViewpoint(true);
    mCamera->setPosition(vp.position);
    mCamera->pitch(90); // Quake uses X/Y horizon, Z up
    mCamera->rotate(vp.orientation);
    // Don't yaw along variable axis, causes leaning
    mCamera->setFixedYawAxis(true, Vector3::UNIT_Z);
}
```

在创建场景的过程中，首先通过场景管理器的 setWorldGeometry 方法载入地图。

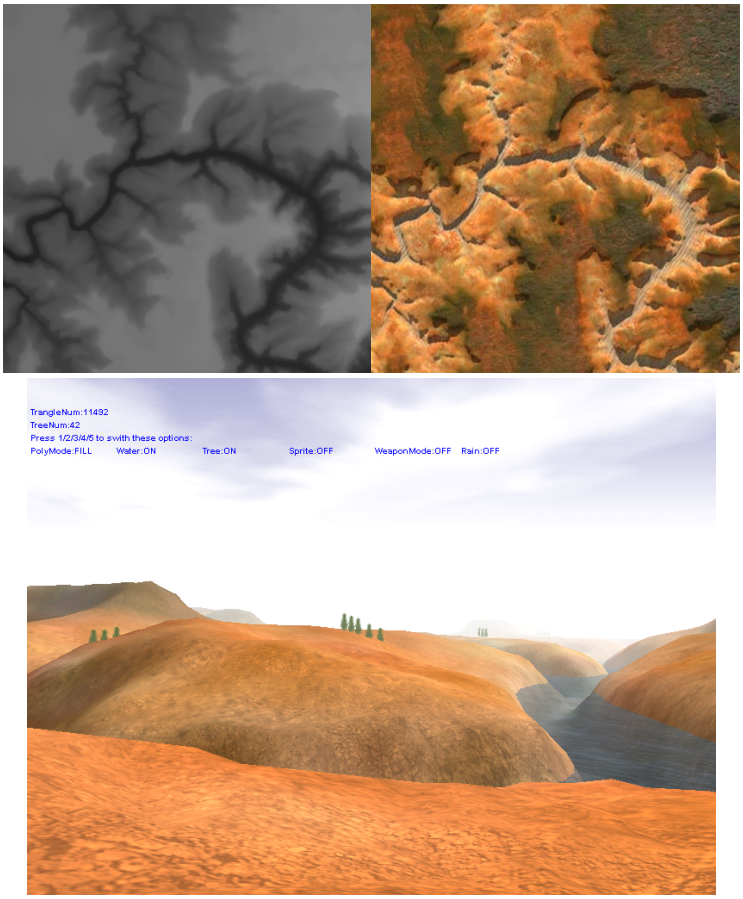
接下来是设置摄象机。QUAKE 地图中都有预先设置好的起始点和起始方向，通过场景管理器的 getSuggestedViewpoint 方法可以获取到这个 ViewPoint，而后再将摄象机定位到这个点上。由于 QUAKE 是以 XY 轴构成的平面为水平面，Z 轴为上方向，所以必须将摄象机用 mCamera->pitch(90)语句调整 90 度。mCamera->rotate(vp.orientation);一句的意思是将摄象机调整到起始方向。最后将摄象机的旋转轴固定，因为人只能头向上走路，不能象飞机那样 360 度全空间翻滚。因为在 QUAKE 中 Z 轴向上，所以通过 mCamera->setFixedYawAxis(true, Vector3::UNIT_Z);语句实现。

室外场景

Demo_Terrain 工程是 OGRE 自带的室外场景的例子，打开该工程查看代码。

OGRE 在其引擎提供的 Plugin_OctreeSceneManager.dll 插件中提供了通过八叉树算法实现的 ST_EXTERIOR_CLOSE 室外封闭场景管理器。OGRE 引擎在初始化的时候会载入 Plugins.cfg 中指定的全部插件，只要在 Plugins.cfg 中包含 Plugin_OctreeSceneManager.dll 插件就可以使用了。

室外场景的主要内容是起伏的地形。表现地形起伏的一般方法是通过一个灰度图来表达场景中每一块土地的高度，颜色浅为高，颜色深为低。引擎读取这个灰度图，并根据每个像素的颜色值画出高低起伏的地形网格。再用另外的彩色图作为地表纹理铺在地形网格上，就实现了室外场景。



除此之外，为了防止地表纹理放大造成的失真，还需要一个代表地面细节的可拼接的细节纹理图。地形显示的时候应该还可以设置缩放因子。OGRE 将这些地形显示需要的图片和属性都放在 terrain.cfg 文件中设置。请查看 terrain.cfg。

在程序中实现室外场景漫游很简单。首先需要选择室外场景管理器，代码如下：

```
virtual void chooseSceneManager(void)
{
    // Get the SceneManager, in this case a generic one
    mSceneMgr = mRoot->getSceneManager( ST_EXTERIOR_CLOSE );
}
```

选择好室外场景管理器后，通过该场景管理器创建场景就可以了。

```
void createScene(void)
{
    // Set ambient light
    mSceneMgr->setAmbientLight(ColourValue(0.5, 0.5, 0.5));
```

```
        // Create a light
        Light* l = mSceneMgr->createLight("MainLight");
        // Accept default settings: point light, white diffuse, just set position
        // NB I could attach the light to a SceneNode if I wanted it to move automatically with
        //    other objects, but I don't
        l->setPosition(20,80,50);

        mSceneMgr -> setWorldGeometry( "terrain.cfg" );
        mSceneMgr->setFog( FOG_EXP2, ColourValue::White, .008, 0,    250 );

        mRoot -> showDebugOverlay( true );
    }
```

以上的 createScene 函数首先设置环境光，又创建了一个点光源。通过 mSceneMgr -> setWorldGeometry("terrain.cfg");语句载入 terrain.cfg 配置文件，解析它，同时创建地形。场景管理器的 setFog 函数可以设置雾化效果。

尽管本例使用的是室外场景管理器，我们依然可以用熟悉的方法向场景中加入 Entity 等场景元素，试着向程序中加入如下代码并查看效果。

```
        Entity *ent = mSceneMgr->createEntity("head", "ogrehead.mesh");
        // Set material loaded from Example.material
        ent->setMaterialName("Examples/EnvMappedRustySteel");
        // Add entity to the root scene node
        mSceneMgr->getRootSceneNode()->createChild()->attachObject(ent);
```

这部分代码向场景中加入了一个食人魔。从这里可以看到，场景管理器的 setWorldGeometry 方法会将固定不变的地形创建好，我们还可以很方便的向场景中加入其它可运动的物体（汽车、动物等），从而实现一个近乎真实的野外世界。

附属工具

MilkShape 导出插件

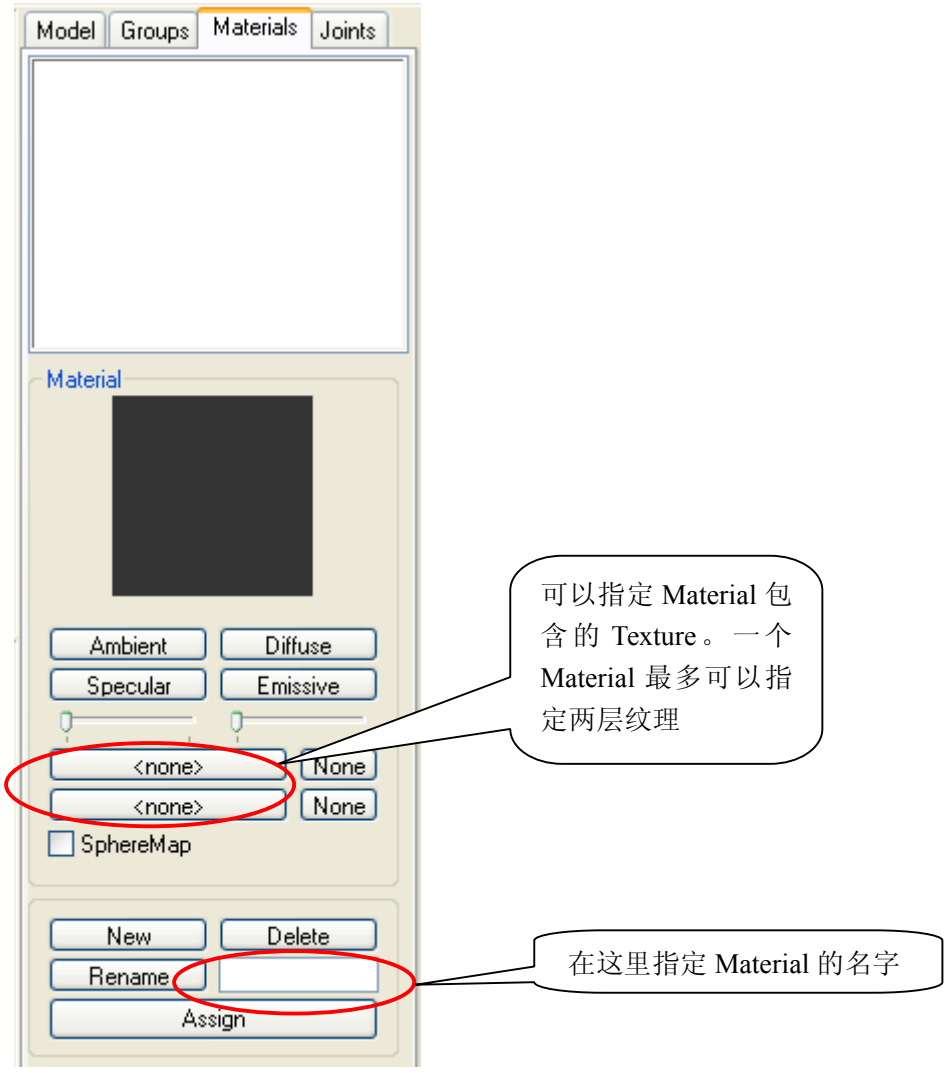
用 MilkShape 导出.mesh 的步骤

1，导入模型

“File” -> “Import” -> “Autodesk 3DS”，然后选择要导入的模型。注意，这里我们选择的是 Autodesk 3DS，它指明要导入的是用 Audodesk 公司的产品 3D Studio 所做的模型，其后缀名是.3ds。我们也可以选择其它格式的模型文件。

2，指定模型的 Material

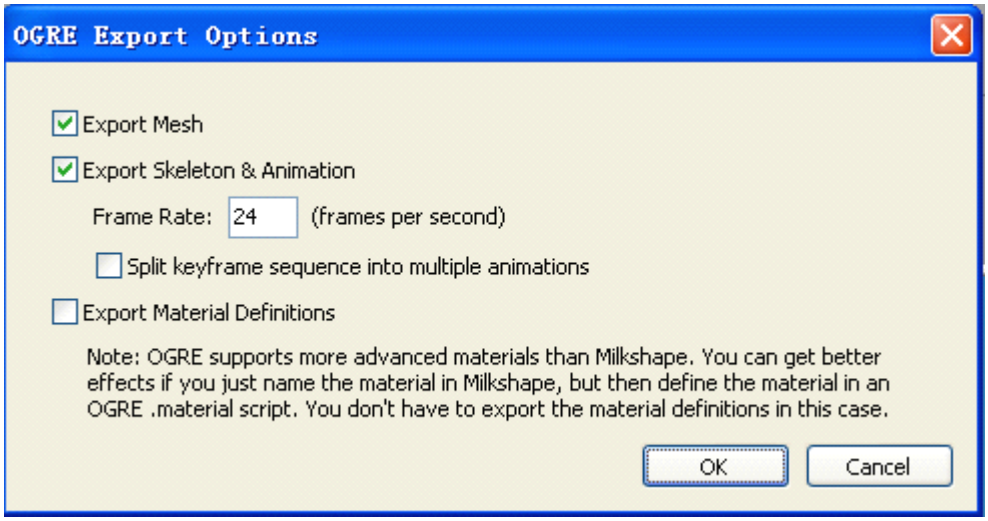
在 Milkshape 中可以设置模型的 Material（这里 Material 的概念比 Texture 更广泛，它可以包含两层 Texture，可以包括纹理坐标，材料属性（Ambient，Specular，Diffuse）等信息）。下图是 Milkshape 的 Material 工具栏，我们可以在这里设置 Material 的属性：



3，将模型导出成 OGRE 所支持的.mesh 格式

要想让 Milkshape 支持导出 OGRE 的模型格式.mesh，首先要把 OGRE 提供的 msOGREExporter.dll 与 OgreMain.dll 拷备到 Milkshape 的工作目录中。

接下来，选择“File” -> “Export” -> “OGRE Mesh/Skeleton…”，这时会弹出一个对话框：



通过这个对话框我们可以设置导出以后 OGRE 模型的属性。

OGRE 提供了较之 Milkshape 更高级的 Material。如果你想获得 OGRE Materialr 的功能，你只需在 Milkshape 中指定 Material 的名字，然后定义自己的.material 脚本，最后把在 Milkshape 中指定的名字做为.material 脚本中 Material 的名字进行设置就可以了。反之，如果你觉得 Milkshpae 的 Material 已经够用，那么就选中最后一个复选框“Export Material Definitions”，从而直接用 Milkshape 的 Material 定义。

如果模型支持骨骼动画则选中“Export Skeleton & Animation”复选框，还可以对动画做一些设置。

3d Studio Max 导出插件

备注

OGRE 相关网站：

1，OGRE homepage：ogre.sourceforge.net
OGRE 的主页，OGRE 更新动向、OGRE Roadmap 值得一看。

2，OGRE in sourceforge：www.sourceforge.net
OGRE 的 CVS 仓库。

3，OGRE Forum：ogre.sourceforge.net/phpBB2/
OGRE 论坛，OGRE 的作者 Steve Streeting 几乎有问必答！

4，Jeff Leigh's Ogre Projects：www.enygmaarts.com/Ogre/
有 Jeff 用 OGRE 做的场景编辑器，很酷！

5，Pong：emotion.sourceforge.net

用 OGRE 开发的弹球游戏，正在开发中。

6，BadCamel：badcamel.sourceforge.net

用 OGRE 开发的扑克游戏。

7，DIE：die.sourceforge.net

用 OGRE 做为图形渲染引擎，用 ODE（开发动力学引擎，ode.sourceforge.net）做为物理引擎开发的大脚本游戏。

8，我们的网站：还没有传到网上，打算以 OGRE 为网站的特色：）

感谢

感谢以下网友的帮助：
skull_wang
gameart
sybase8
wizard_kai
wu_hao
fannyfish