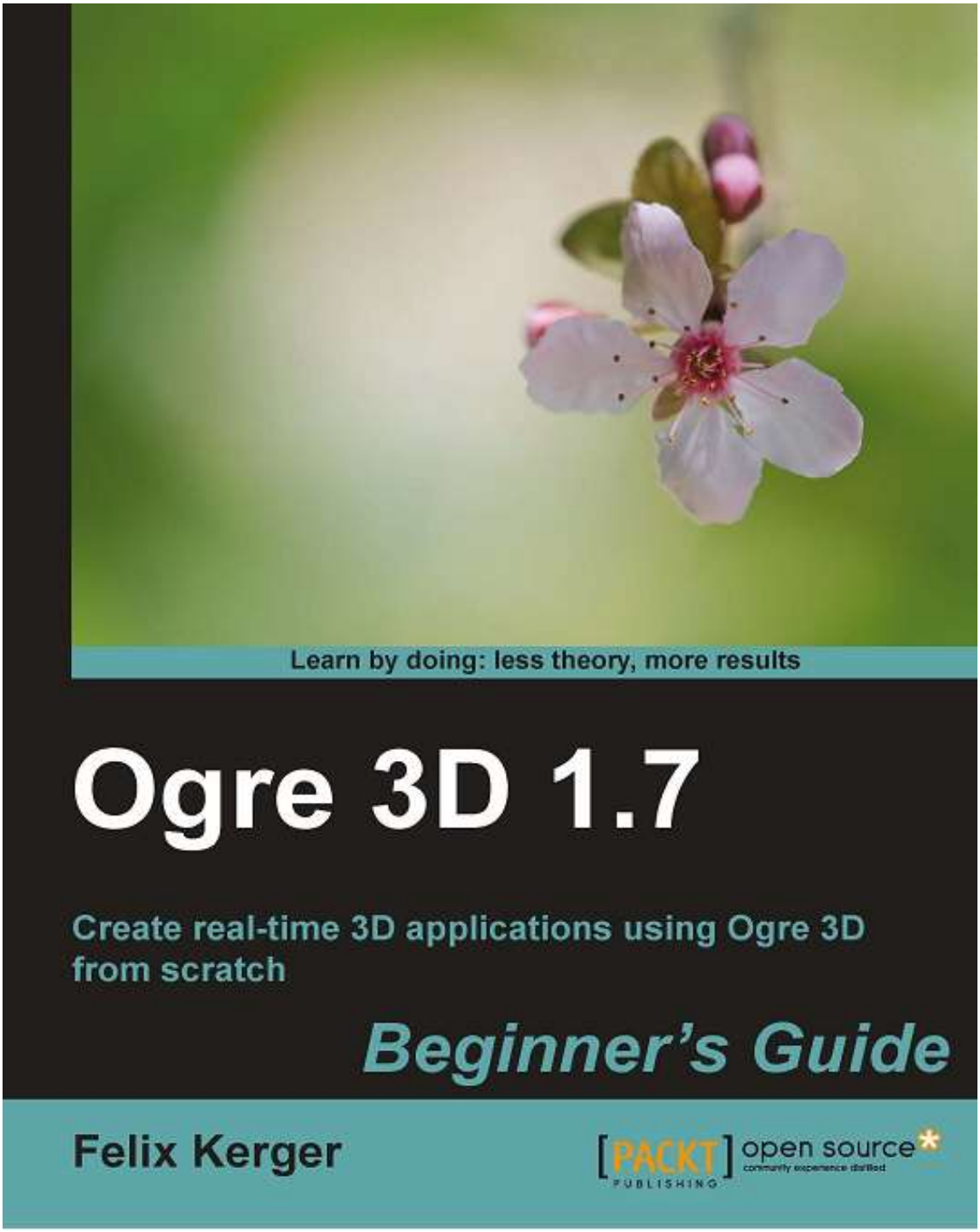

Learn by doing: less theory, more results
Ogre 3D 1.7
Create real-time 3D applications using Ogre 3D from scratch
Beginner's Guide
Felix Kerger
[PACKT] open source
PUBLISHING

关于作者

Felix Kerger 是达姆施塔特科技大学的一名主修计算机科学的学生。5 年多的时间里他一直在采用 Ogre 3D 开发三维实时应用程序。他曾在不同的会议上发表一些关于软件开发和三维实时应用程序的讲话，而且作为一名研究助理在弗劳恩霍夫计算机制图学研究所工作了 3 年。他也是一名自由撰稿人，每年都会为游戏开发者欧洲代表大会撰写报告。

我要感谢以下各位，没有他们，这本书不可能写成：Steve Streeting，感谢他为 Ogre 3D 投入那么多时间而且创作了堪称最好的软件之一，也是我很荣幸用到的软件；我曾经的老师 Oppel 和 Michel 女士，感谢她们的帮助，使我写成了为本书的精髓思想打基础的报告；我的父母，感谢他们成为我灵感和动力的不竭源泉；Gregory Junker 和 Manuel Bua，我的技术评审——他们不断提出建议帮助我，使本书得到了很大提高；当然，还有 Packt 团队，感谢他们不断地帮助和提议。

关于评审

Manuel Bua 是一位来自意大利特兰托的软件和解决方案设计师。他有超过 17 年的经验并且参加过很多大型和小型的软件项目，包括设计和实现。他的背景和经历从软件开发延伸到逆向工程，包括桌面和移动平台；多线程，并行和大规模并行高性能嵌入式计算也大大激发了他的兴趣，以及计算摄影学和游戏开发。

2007 年，他在法国巴黎参加了 Jooce 研究发展部，担任总设计师的职务，设计优化他们内部基于 ActionScript 编程语言的虚拟桌面平台，连接世界各地成百上千的人；在此期间，他还设计实现了一款合成窗口管理器，管理窗口过渡和效果，就像著名的“沃布利视窗”，初步被在 Linux 桌面系统内(rocking!)的 Compiz 引入。

他热爱开放标准和开源文化。他对学习和分享知识的渴望引导他奉献于各种项目，例如 Ogre 本身；他设计编程了原始的外核心实现模式，也就是今天所知的“排字框架”，提供了最初的洞察力和高层次的概念，为进一步的研究，工作和进步打下了基础

他目前在 F4F 创意工厂工作，一个总部设在特兰托阿科公司的关于创作灵感的网站和广告策划机构，担任着解决方案设计师，软件工程师和系统管理员的职务。

序言

创建 3D 影像世界是一个有趣并且充满挑战的问题，但其研究的效果很有回报性而且其探索的过程会相当有趣。本书会向你展示如何用 Ogre3D 创造出属于你自己的场景和世界。Ogre 3D 是最大的开放源码三维渲染引擎中的其中之一，并且能够令使用它的人自由地创造和与他们的场景进行交互。

这本书没有涵盖关于 Ogre 3D 的所有细节，而是提供了一个详实的介绍，作为读者的你，根据此书，能够开始自己使用 Ogre 3D。读完本书后，你能够查找本书没有包含的所需信息和复杂的技巧。

本书提要

章节 1，安装 Ogre 3D， 展示如何获取和安装 Ogre 3D。我们还要创造我们的第一个场景并且开始学习 Ogre 3D 的内容。

章节 2，Ogre 场景图，为我们介绍了场景图的概念以及如何运用它描述 3D 场景。

章节 3，相机，光与影，添加光与影到我们的场景中并尝试不同的相机设置。

章节 4，获取用户输入并使用 Frame 监听器，采集用户输入，并让他们相互作用。

章节 5，Ogre 3D 动画模型，采用动画加入更多的互动和现实感来强化我们的场景。

章节 6，场景管理，向我们介绍组织 3D 场景的不同理念和这些选择会有怎样的含义。

章节 7，Ogre 3D 材质，向我们展示如何运用材料和渲染程序来增添细节和灵活性。

章节 8，合成器框架，向我们展示如何添加后期效果来改变我们的场景外貌。

章节 9，Ogre 3D 启动过程，向我们展示了不使用实例的帮助时如何运用 Ogre 3D。

章节 10，粒子系统和 Ogre3D 的延伸。给出了一些运用 Ogre3D 能达到的更加先进的技术和前景。

读本书所需基础

为学习本书，你需要很好得掌握 C++语言，熟悉如何运用 C++ 编写应用程序。当然，你需要一个编辑器来编辑示例应用程序。本书使用 Visual Studio 作为参考，但其他编辑器也是可以的。你的电脑需要装有 3D 加速的显卡。如果你的显卡支持 DirectX 9.0 就更好了，因为 Ogre3D 是一个开源软件，我们要在章节 1 中下载它。因此，你的电脑不需要事先安装 Ogre 3D。

读者

如果你曾想用 Ogre 3D 开发 3D 应用程序，这本示例驱动的书能够使你梦想成真。你需要掌握 C++来学习本书中的例题。本书是一本示例驱动的 Ogre3D 简介。每个例题都会显示一些新的特征，你将一步步地学习用 Ogre 3D 制造出有着不同效果的复杂场景。看完几个关于一个主题的例题后，会有一个练习部分，在这个部分，你将要迎接自己解决问题的挑战。

惯例

为了把不同种类的信息区分开来，您将会在本书中发现一系列的文本样式。这里有这些样式的一些列子，以及它们含义的解释。

文中的编码显示如下：Delete all the old code in createScene()， except for the plane-related code. 一个代码块的示例如下：

```
void MyFragmentShader2(float2 uv: TEXCOORD0，out float4 color : COLOR，
                                                    uniform sampler2D texture)
```

新术语和重要词语用粗体显示。举例说明，你在屏幕上，菜单里或对话框里看到的字，在文中是这样显示的：“**点击 OK 启动一个应用程序。**”

关于译者

前言部分	琳琳
第一章	程光曦微
第二章—第九章	呢喃的歌声
第十章	Fss

翻译声明：

本人仅仅英语六级水平，因为喜欢翻译和喜欢 Ogre，翻译以下文字，如果哪里有错误或不足还望大虾们见谅，如果您发现有什么错误，请把问题发送到 whistleofmysong@gmail.com 或者在我的博客 www.singmelody.com 留言，我将尽快的改正。这对他人是个帮助，对自己也是个鼓励~^_^

相关法律

《中华人民共和国著作权法》

第二节 著作权归属

第十二条 改编、翻译、注释、整理已有作品而产生的作品，其著作权由改编、翻译、注释、整理人享有，但行使著作权时，不得侵犯原作品的著作权。

第四节 权利的限制

第二十二条 在下列情况下使用作品，可以不经著作权人许可，不向其支付报酬，但应当指明作者姓名、作品名称，并且不得侵犯著作权人依照本法享有的其他权利：

（六）为学校课堂教学或者科学研究，翻译或者少量复制已经发表的作品，供教学或者科研人员使用，但不得出版发行；

相关资源：

Ogre 3D wiki：http://www.ogre3d.org/tikiwiki/
Ogre 3D 中文 wiki: http://ogre3d.cn
本书源代码地址: http://download.csdn.net/detail/chinarpgmaker/4031545

相关建议：

建议大家结合着 Ogre 官方的 wiki 一起看，虽然是英文，但是可以原滋原味的理解清除 Ogre 到底是怎么回事。还有一本非常好的 Ogre 的书 《OGRE 3D 程序设计》，这本书详

细的讲解了 Ogre 的原理部分。此书重点在于入门，应用和基础概念的讲解。重点的章节为第六章，第七章和第九章

感想与感悟：

"When I was a young man， I had liberty， but I did not see it. I had time， but I did not know it. And I had love， but I did not feel it. Many decades would pass before I understood the meaning of all three. And now， the twilight of my life， this understanding has passed into contentment.

Love， liberty， and time: once so disposable， are the fuels that drive me forward. And love， most especially， mio caro. For you， our children， our brothers and sisters. And for the vast and wonderful world that gave us life， and keeps us guessing. Endless affection， mio Sofia.

Forever yours，

Ezio Auditore."

整理说明：

此中文版翻译原来是中英对照的，没有排版，看起来比较吃力。所以我整理了一份纯中文版的，重新进行了排版以方便阅读，整理的过程中也对部分言语不通的地方做了少许的修正。整理后的版本发布在 http://www.adintr.com/media/books/ogre_3d_1.7_beginner_guide.pdf。如对此整理版本有何疑问，可以通过发送电子邮件到 *webmaster@adintr.com* 或者通过网站 *http://www.adintr.com* 上的留言板进行反馈。如对翻译有疑问，请按上面翻译声明进行反馈。

第一章 安装 Ogre 3D 引擎

想要学习并且使用 Ogre，就要先下载并安装它.

在这章，我们将会学习到：
* 下载并安装 Ogre 3D 引擎
* 配置好 Ogre 3D 引擎的开发环境
* 用 Ogre 3D 引擎构建第一个场景

那我们就开搞了。

1.1 下载并安装 Ogre 3D 引擎

我们要干的第一步就是安装并配置 Ogre 3D 引擎

实践时刻——下载并安装 Ogre 3D 引擎

为了后面的使用，现在我们就要下载安装 Ogre 3D SDK 了

1. 打开 http://www.ogre3d.org/download/sdk. 的链接

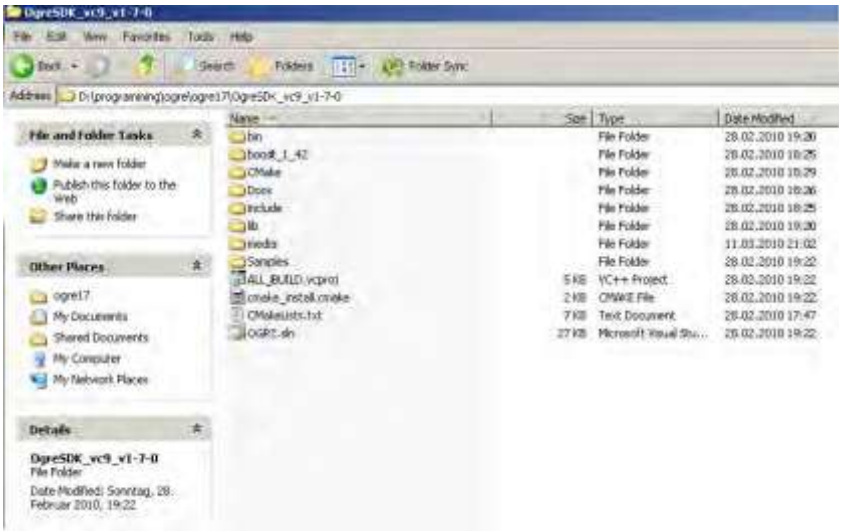
2. 下载合适的安装包。如果你对要下载哪个正确的安装包而需要帮助时，那就看看后面的“刚刚发生了什么”这一部分

3. 把 SDK 的安装程序复制到你希望把 OgreSDK 放置的文件夹里

4. 双击安装程序，这会启动一个自解压缩程序

5. 现在你应该得到一个名字类似于 OgreSDK_vc9_v1-7-1 的文件夹

6. 打开该文件夹。文件夹的内容应该像下面的截图



刚刚发生了什么？

我们刚刚只是下载了一个适合我们操作系统的 Ogre 3D SDK。Ogre 3D 引擎是一个跨平台的渲染引擎，所以对于不同的操作系统就有不同的开发包。下载完 Ogre 3D SDK 我们解压了它。

不同版本的 Ogre 3D SDK

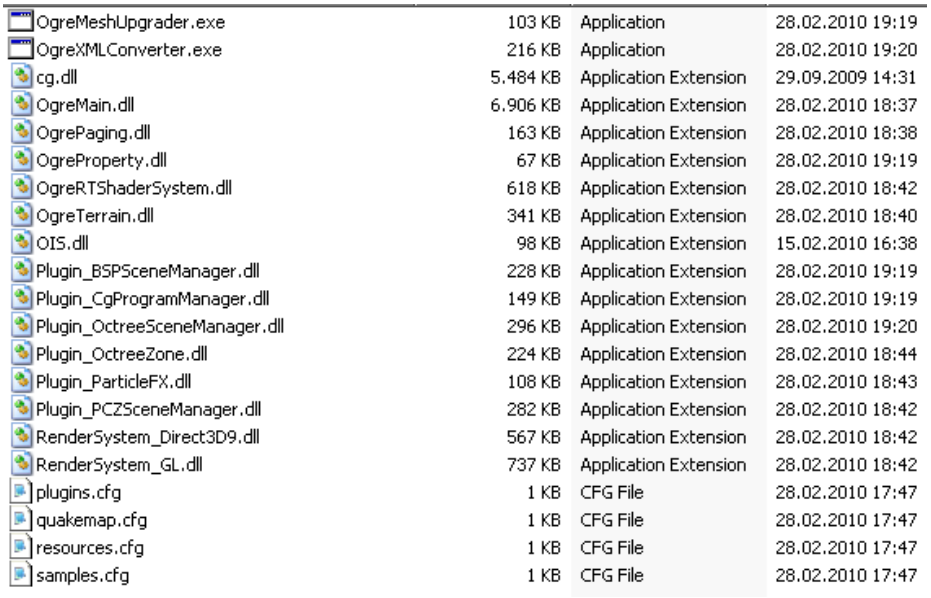
Ogre 支持多种平台，正因为如此，有很多种不同的开发包供我们下载。Ogre 3D 在 Windows 有好几个版本，有一个支持 MacOSX 的版本还有一个支持 Ubuntu 的版本。而且还提供了支持 MinGW 和 iPhone 的开发包。如果你愿意的话，你还可以下载 Ogre 的源码然后自己手动编译 Ogre。这一章我们主要关注于预先编译好的 Windows 的 SDK 和如何配置你的开发环境。如果需要关于其他的操作系统的，你可以参考 Ogre 3D Wiki，网址为：http://www.ogre3d.org/wiki。Wiki 包含了对很多不同的平台下的 Ogre 开发环境的配置的教程。这本书的剩余部分和使用的平台是完全独立的，所以如果你想使用其他的开发平台，随你便啦。它不会影响到本书的内容和对编译系统的配置与约定。

探索 SDK

在我们编译 SDK 里面的示例之前，让我们先来看看 SDK。我们来看看 Windows 操作系统上的 SDK 结构。在 Linux 和 MacOS 上可能会有所不同。首先，我们打开 bin 这个文件夹。这里面会有两个文件夹，即 debug 和 release 文件夹。对于 lib 文件夹里面也是如此。这里面的原因是在于 Ogre 3D SDK 对其库文件和动态链接库文件有 debug 和 release 两种编译方式。这就使得我们可以在开发过程中使用 debug 模式来调试我们的项目。完成项目后，可以使用 release 模式来编译得到完整的 Ogre 3D 程序。

不论打开 debug 或者 release 文件夹，我们都可以看到有很多的 dll 文件，一些 cfg 文件

还有两个可执行文件（exe）。可执行文件是为了把 Ogre 升级到一个新的版本，所以在此对我们来说没有啥用处。



OgreMeshUpgrader.exe	103 KB	Application	28.02.2010 19:19
OgreXMLConverter.exe	216 KB	Application	28.02.2010 19:20
cg.dll	5.484 KB	Application Extension	29.09.2009 14:31
OgreMain.dll	6.906 KB	Application Extension	28.02.2010 18:37
OgrePaging.dll	163 KB	Application Extension	28.02.2010 18:38
OgreProperty.dll	67 KB	Application Extension	28.02.2010 19:19
OgreRTShaderSystem.dll	618 KB	Application Extension	28.02.2010 18:42
OgreTerrain.dll	341 KB	Application Extension	28.02.2010 18:40
OIS.dll	98 KB	Application Extension	15.02.2010 16:38
Plugin_BSPSceneManager.dll	228 KB	Application Extension	28.02.2010 19:19
Plugin_CgProgramManager.dll	149 KB	Application Extension	28.02.2010 19:19
Plugin_OctreeSceneManager.dll	296 KB	Application Extension	28.02.2010 19:20
Plugin_OctreeZone.dll	224 KB	Application Extension	28.02.2010 18:44
Plugin_ParticleFX.dll	108 KB	Application Extension	28.02.2010 18:43
Plugin_PCZSceneManager.dll	282 KB	Application Extension	28.02.2010 18:42
RenderSystem_Direct3D9.dll	567 KB	Application Extension	28.02.2010 18:42
RenderSystem_GL.dll	737 KB	Application Extension	28.02.2010 18:42
plugins.cfg	1 KB	CFG File	28.02.2010 17:47
quakemap.cfg	1 KB	CFG File	28.02.2010 17:47
resources.cfg	1 KB	CFG File	28.02.2010 17:47
samples.cfg	1 KB	CFG File	28.02.2010 17:47

OgreMain.dll 是最重要的 DLL 文件。后面编译的 Ogre 3D 程序都会用到它。所有名字以 Plugin_开头的 DLL 文件是 Ogre 3D 程序可以使用的插件。Ogre 3D 插件是使用 Ogre 3D 提供的接口来给 Ogre 3D 添加新功能的动态链接库文件。插件可以提供任何东东，但是它常常是用来添加一些特性例如更好的粒子系统或者新的场景管理器。这些东西后面会谈到的。Ogre 3D 社区提供了很多的插件，大部分可以在 wiki 中找到。SDK 中只是包含了最近常用的插件。在本书后面，我们会学习如何去使用它们。名字以 RenderSystem_开头的 DLL 文件，不要惊讶啊，是为了封装起来针对于 Ogre 3D 所支持不同的渲染系统。在这里所指的是 Direct3D9 和 OpenGL。除了这两个系统，Ogre 3D 还支持 Direct3D10、Direct3D11 和 OpenGL ES（支持嵌入式系统的 OpenGL）渲染系统。

除了可执行文件和 DLL 文件，还有一些 cfg 文件。所谓的 cfg 文件就是 Ogre 3D 程序可以加载的配置文件。Plugins.cfg 列出了 Ogre 3D 程序启动时所要加载的全部的插件。这些通常是 Direct3D 和 OpenGL 渲染系统还有一些附加的场景管理系统插件。当加载 Quake3 格式的地图时需要 quakemap.cfg 这个配置文件。我们用不上这个文件，但是有个例子用得上。

resources.cfg 包含了所有的资源的列表，例如一个 3D 网格、纹理或者动画，这些都是 Ogre 3D 启动时需要加载的。Ogre 3D 程序可以从文件系统或者一个 ZIP 压缩文件中加载资源。我们如果看看这个 resources.cfg 文件内容就可以得到以下几行：

```
Zip=../../media/packs/SdkTrays.zip
FileSystem=../../media/thumbnails
```

ZIP=意味着那些资源文件是存在 ZIP 压缩包里的，FileSystem=意味着要加载一个文件

夹里的内容。resources.cfg 使得加载新的资源和改变资源的路径非常容易，所以它常常被用来加载资源，特别是被 Ogre 的例子程序所使用。来说一下示例程序，文件夹中的最后一个 cfg 文件是 sample.cfg。我们自己用不上这个文件。其实它包含着一个 SampleBrowser（示例浏览器）程序所需加载的所有 Ogre 示例列表。但是我们还没搞到 SampleBrowser，所以就要搞一个出来。

1.2 Ogre 3D 示例程序

Ogre 3D SDK 附带了很多的示例，这些示例展示了 Ogre 3D 所拥有的不同种类的渲染效果和手法。在我们开始编写我们自己的程序之前，为了加深 Ogre 程序的功能我们先来看看这些示例。

实践时刻——构建 Ogre 3D 示例程序

为了先看看 Ogre 3D 到底能干嘛，我们会构建示例程序并且看看它们。

1. 找到 Ogre3D 的文件夹
2. 打开 Ogre3d.sln 解决问题方案文件
3. 在解决问题方案上右键选择“构建解决问题方案”
4. Visual Studio 会构建示例程序，这会消耗掉一些时间，所以在编译没有完成之前最好是泡上杯茶喝喝
5. 若一切顺利，找到 Ogre3D/bin 文件夹
6. 运行 SampleBrowser.exe.
7. 然后你应该看到如下给力的结果：

8.试试运行不同的示例程序来看看 Ogre 3D 带来的精彩的特效。

刚刚发生了什么？

我们使用自己的 Ogre 3D SDK 构建出 Ogre 3D 示例。在此之后，我们会有一个 Ogre 3D 开发的副本

突击测验——示例程序里展示了哪些后期效果

列举至少 5 种在示例程序中展示了的不同后期效果

a. 模糊，玻璃特效，旧电视感觉，黑白效果和倒置
b. 模糊，玻璃特效，旧显示器感觉，黑白效果和倒置
c. 模糊，玻璃特效，旧电视感觉，颜色和倒置

1.3 第一个 Ogre 3D 程序

在这一部分，我们会创建一个只有一个 3D 模型的 Ogre 3D 程序。

实践时刻——创建项目并配置 IDE（集成开发环境）

由于用到了其他的一些库，在使用 Ogre 3D SDK 之前我们需要配置 IDE

1. 新建一个空工程
2. 在工程里新建一个文件，命名为 main.cpp
3. 添加一个 main 方法：

```
int main (void)
{
    return 0;
}
```

4. 在该文件开头部分包含上 ExampleApplication.h 文件：

```
#include "Ogre\ExampleApplication.h"
```

5. 添加“你的 Ogre SDK 路径\include\”到你的项目 include path（头文件路径）
6. 添加“你的 Ogre SDK 路径\boost_1_42\”到你的项目 include path（头文件路径）
7. 添加“你的 Ogre SDK 路径\boost_1_42\lib\”到你的项目 lib path（库文件路径）
8. 向 main.cpp 里添加一个类：

```
class Example1 : public ExampleApplication
{
public:
    void createScene()
    {
    }
};
```

9. 把下面的代码添加到你的 main 函数里：

```
Example1 app;
app.go();
```

10. 添加“你的 Ogre SDK 路径\ lib\debug\”到你的项目 lib path（库文件路径）
11. 添加 OgreMain_d.lib 到需要的链接库（工程属性的 Linker->Input）里
12. 添加 OIS_d.lib 到需要的链接库（工程属性的 Linker->Input）里
13. 编译此项目
14. 设置项目程序的工作空间（working directory）为“你的 Ogre SDK 路径\bin\debug”
15. 运行该程序，你应该能够看到 Ogre 3D setup 对话框

16. 按下 OK 按钮启动应用程序。你会看到一个黑色的窗口。按下 Esc 键退出程序。

刚刚发生了什么？

我们创建了第一个 Ogre 3D 程序。为了编译它，我们需要设置好不同的头文件路径和库文件路径以便于编译器能够找到它们。

在第 5 和第 6 步，我们在构建环境中添加了两个头文件的路径。第一个路径是 Ogre 3D SDK 的头文件文件夹，它包含着所有 Ogre 3D 和 OIS（OIS 是 Object Oriented Input System 的简称，意为“面向对象的输入系统”，ExampleApplication 使用 OIS 监视用户的输入）的头文件。OIS 不是 Ogre 3D 中的一部分；它是一个独立的项目并且有一个不同的开发团队在开发。它出现在 Ogre 3D 是因为 ExampleApplication 使用了它，所以用户不需要去下载它的依赖文件。 ExampleApplication.h 也在那个文件夹中。由于 Ogre 3D 提供线程支持，所以我们需要把 boost 文件夹添加入头文件路径。否则的话，我们用 Ogre 3D 引擎什么程序也编译不了。如果需要的话，Ogre 3D 可以直接从源码编译，那样的话可以取消线程支持也就取消掉了 boost 文件夹的需求。当使用 boost 的时候，编译器需要链接到 boost 的库文件。所以我们添加了 boost 的库文件文件夹到 lib path（见第 7 步）。

在第 10 步中，我们把“你的 Ogre SDK 路径\ lib\debug\”添加到了库文件路径之中，像前面所说的那样，Ogre 3D 有 debug 库和 release 库。在这里我们使用 debug 库，因为如果出现错误，debug 库提供调试支持。当想使用 release 库的时候，要把“lib\debug”改为“\lib\release”。对于第 11 步和第 12 步也是如此。在那两步中我们添加了 OgreMain_d.lib 和 OIS_d.lib 作为需要链接的库文件。当想使用 release 版本的时候，要改成 OgreMain.lib 和 OIS.lib。OgreMain.lib 和 OgreMain_d.lib 文件包含着 Ogre 3D 程序的接口信息和怎么样加载 OgreMain.dll 或 OgreMain_d.dll。注意 OIS.lib 或 OIS_d.lib 对于输入系统也是如此——他们加载 OIS_d.dll 或 OIS.dll。所以我们动态的链接 Ogre 3D 和 OIS，这样使得我们可以切换 DLL 而不需要重新编译程序，只要接口和库没有变化而且 DLL 使用相同的运行库版本。这也就要求程序时刻需要加载 DLL，所以要确保程序能够找到那些 DLL。这也就是我们为啥要在第 14 步设置工作空间。还有一个原因会在后面的部分澄清。

ExampleApplication（Ogre SDK 中的一个类）

我们创建了一个新的类，Example1，它继承于 ExampleApplication。ExampleApplication 是 Ogre 3D SDK 提供的一个类，它是为了使学习 Ogre 3D 简单一些而在 Ogre 3D 上附加了一个抽象层。ExampleApplication 为我们开始学习 Ogre 提供了帮助，它可以加载不同的模型，而且声明了一个可以在场景中四处游览的简单 camera（摄像机）。使用 ExampleApplication 需要继承于它并且重载虚函数 createScene()。我们会使用 ExampleApplication 类，这样的话可以省下不少的时间。等到对 Ogre 3D 有了很好的了解之后，我们会用自己的代码一点点的替换掉 ExampleApplication。

在 main 函数里，我们实例化了一个新的类并且调用了 go()函数来启动应用程序并加载 Ogre 3D。在启动时，Ogre 3D 加载 3 个配置文件——Ogre.cfg、plugins.cfg 和 resources.cfg。如果使用的是 debug 模式，所以要每个文件名字后面要加上“_d”。这个非常有用因为对 debug 和 release 有不同的配置文件。Ogre.cfg 包含着在 setup 对话框中选择的设置，所以每次程序

启动它加载相同的配置。plugins.cfg 包含 Ogre 需要加载的插件列表。最重要的插件是渲染系统插件。它是 Ogre 和 OpenGL 或 DirectX 渲染场景的接口。resources.cfg 包含着 Ogre 启动时需要加载的资源列表。Ogre 3D SDK 附带了很多的模型和材质，本书会用到这些，resources.cfg 指出了他们的路径。如果你深入了解 resources.cfg，你会发现这个文件中的路径都是相对路径。这也是我们需要设定工作空间的另一个原因。 】

突击测验——要链接哪个库

1.当使用 Ogre 3D 的 release 模式的时候，你认为要链接哪些库？

a. OgreD3DRenderSystem.lib
b. OgreMain.lib
c. OIS.lib

2.当想使用 Ogre 3D 的 debug 模式的时候，你认为该做哪些改变？

a. 在库文件名后面加上“_debug”
b. 在文件扩展名后面加上“_d”
c. 在库文件名后面加上“_d”

加载第一个 3D 模型

加载一个模型很容易。只需要加上下面两行代码

1. 把下面两行代码加入到空的 *createScene()* 方法中：

```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity"， "Sinbad.mesh");
mSceneMgr->getRootSceneNode()->attachObject(ent);
```

2. 再次编译你的程序
3. 启动程序。你会看到一个小的绿色的东东。
4. 使用鼠标四处浏览场景，使用“WASD”移动视角直到看着感觉绿色的东东在视角中比较合适
5. 关闭程序

刚刚发生了什么？

使用 *mSceneMgr->createEntity("MyEntity"，"Sinbad.mesh");*，我们告诉了 Ogre 我们想创建一个 Sinbad.mesh 模型的实例。 mSceneMgr 是 Ogre 3D 中指向 SceneManager（场景管理器）的指针，这个指针由 ExampleApplication 为我们创建。为了创建一个新的 entity（实体），Ogre 需要知道使用哪个模型文件，而且我们可以给新的实例起一个名字。这个名字是独一无二的非常重要，它不可以被用两次。如果发现了使用两次，Ogre 3D 会抛出一个异常。如果没有起一个名字，Ogre 3D 会自动生成一个。后面我们会详细的讲到。

现在我们有了一个模型的实例，为了使它显示出来，我们需要把它附加在场景之中。添加场景很容易——只要加上下面一句话：

```
mSceneMgr->getRootSceneNode()->attachObject(ent);
```

这样的话，附加实体到场景中以便我们能够看到它。我们所看到的是个水手，是 Ogre 3D 模型的吉祥物。贯穿这本书我们将看到很多次这个模型。

突击测验——ExampleApplication 和怎样显示出一个模型

用自己的话说说怎样加载一个模型并且使其可见。

1.4 总结

我们学习了 Ogre 3D SDK 是如何组织的，哪些库是需要链接的，还有哪些文件夹是需要加入头文件路径。而且，我们瞄了一眼 ExampleApplication 这个类并且学习了如何去使用它。我们加载了一个模型并且显示了出来。明确一下，这一章包含了：

1. 哪些文件对于 Ogre 3D 开发是重要的，它们怎么相互影响，还有它们的作用是啥
2. ExampleApplication 是干嘛的：它是怎样节省工作量的，Ogre 3D 程序启动时干了什么
3. 模型加载：我们学会了使用 createEntity 来创建一个新的模型实例，学会了一种把新实例附加到场景中的办法

在这些关于 Ogre 3D 的介绍之后，我们将在下一章学习 Ogre 3D 怎样管理场景并熟练地操纵场景。

第二章 Ogre 之场景绘图

这章将会介绍给我们场景绘图的一些概念和如何使用函数创造一个复杂的场景。

在这章，我们将会：

* 学习在 3D 空间中三个基本的操作。
* 一个场景绘图是如何被组织的。
* 我们可以操作的不同的 3D 空间。

那么，就让我们开始吧。

2.1 创建一个场景结点。

在上一章中（第一章 创建 Ogre 3D），我们加载了一个 3D 模型并且把它绑定到我们的场景上。现在我们将会学习如何创建一个新的场景结点并如何把 3D 模型绑定到结点上。

实践时刻 — 用 Ogre3D 创建一个场景结点

我们将会使用第一章的代码，修改第一章的代码以创建一个新的场景结点，并且把他绑定到一个场景结点上。我们将会做以下步骤：】

1. 在我们老版本的代码中，我们在 createScene() 函数中存在以下两行。

```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity"， "Sinbad.mesh");
mSceneMgr->getRootSceneNode()->attachObject(ent);
```

2. 用以下代码替换函数中的最后一行。

```
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
```

3. 然后添加以下两行；这两行代码的添加顺序对场景的效果没有影响。

```
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```

4. 编译并且运行应用程序。

5. 当你运行应用程序，你应会看到和你第一章一样的场景。

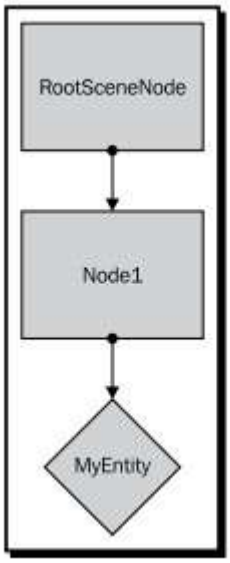
刚刚发生了什么？

我们创建了一个新的命名为 Node1 的场景结点。然后我们添加这个场景结点到根结点。在这之后，我们绑定之前的 3D 模型到我们新创建的结点上面，这样就可以看到效果了。

如何使用场景根结点

调用 mSceneMgr->getRootSceneNode() 函数将会返回场景的根结点。场景根结点是场景管理器的一个成员变量。当我们想要什么显示的时候，我们需要以一种方式把它绑定到场景根结点上或者一个派生类或子类的结点上。简而言之，子结点需要和根结点保持结点与结点之间的联系，否则的话，子结点上的模型就得不到渲染。就像变量名所暗示的那样，场景根结点是场景的根。因此整个场景将会以某种方式绑定到场景的根结点上。Ogre 3D 使用了所谓的场景绘图的方式来组织场景。

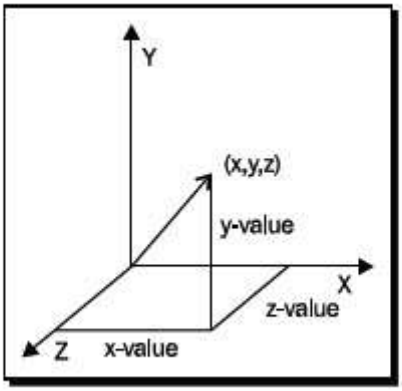
这个绘图的方式就好像一棵树一样。它只有一个根（也就是场景根结点），并且每个结点都可以有子结点。我们在 mSceneMgr->getRootSceneNode()->addChild(node)中使用了这一特性。这里把创建好的场景结点当做场景根结点的子结点。在这之后，我们使用 node->attachObject(ent) 使另一个子节点添加到场景结点。这里，我们添加一个实体到场景结点。我们现在有两种不同的可以添加到场景结点的对象。首先，可以被添加为子结点或者可为自己添加子结点。然后，我们我们创建我们想要渲染的实体。实体不是子结点，它们也不可能有子结点。这些数据对象可以当作是被关联上的结点也可以被认为是树上的叶子。我们稍晚将会学习它们到底是什么和如何使用它们。现在，我们只需要实体。我们当前的场景绘图方式就好像下图一样。



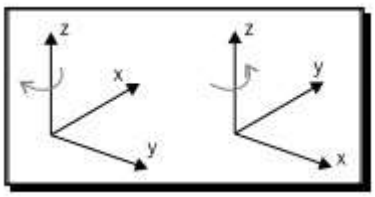
首先我们需要理解的是什么是场景结点和他们是干什么的。一个场景绘图就是用来代替在 3D 空间中不同部分的场景是如何相互关联在一起的。

3D 空间

Ogre 3D 是一个 3D 渲染引擎，所以我们需要一些基本的 3D 概念。在 3D 上最基础的结构就是向量，这个向量中有次序的包含着(x，y，z)。



在 3D 空间的每个位置都可以被欧几里德三维坐标系的一个三维坐标所表示。我们要强调一下在 3D 表示方面有不同的坐标体系。而这些体系之间的不同就是主轴的方向和旋转的正方向。现在主流的有两种体系，左手坐标系和右手坐标系。在下面的图中，我们看到两种体系 — 在左边我们看到的是左手坐标系;在右边我们看到的是右手坐标系。



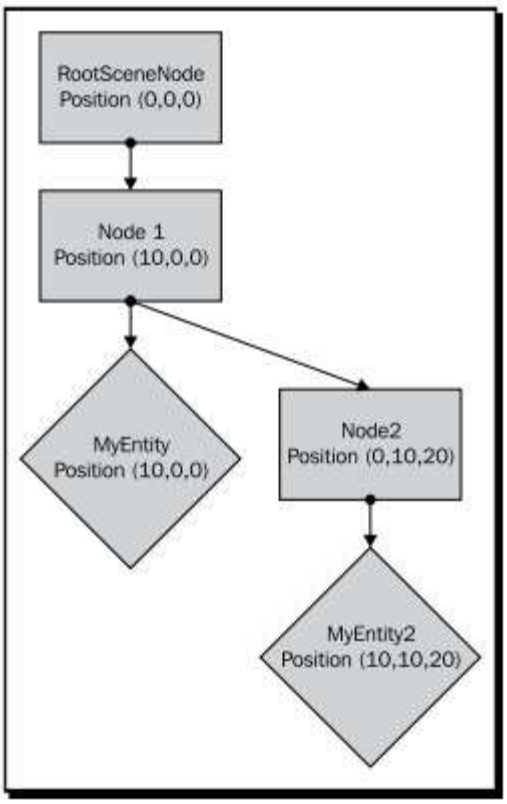
左手和右手坐标系名字的由来是基于他们主轴方向的不同，方向是左手和右手来决定创建的。大拇指是 X 轴，食指是 Y 轴，而中指是 Z 轴。我们需要张开手使大拇指，食指和中指之间保持 90 度角。当使用右手时，我们就可以得到一个右手坐标系。当使用左手时，我们就可以得到一个左手坐标系

Ogre 使用的是右手坐标系，但是旋转坐标系使 X 轴的正半轴指向右并且 X 轴的负半轴指向左。Y 轴(正半轴)指向上，Z 轴（正半轴）垂直于屏幕向外，这被称为 y-up convention。这开始会使我们很不适应。但是我们不久将会在这样的坐标系下学习研究 Ogre。这个网址 http://viz.aset.psu.edu/gho/sem_notes/3d_fundamentals/html/3d_coordinates.html 有能更好展示不同坐标系之间的不同和联系的图片解释。

场景绘图

场景绘图是在图形化编程领域最被广泛使用的概念之一。简单的说，这是存储场景信息的一种方式。我们已经讨论过场景绘图必须有一个根结点而且是树形结构的。但是我们还没有涉及场景绘图中最重要的函数。每个场景结点既有子结点也有可以在 3D 空间变换的函数。这些变换可以说由三方面组成，就是—— 位置变换(position)，旋转变换(rotation)和缩放(scale)变换。坐标点的(x，y，z)，明确的描述了结点在场景中的位置。旋转是使用四元数来储存的，四元数是 3D 空间存贮旋转的一个数学概念，但我们可以认为旋转就是每个坐标轴一个的浮点数，描述了结点以弧度为单位的旋转程度。缩放就比较简单了，同样的，就是它使用了一个（x，y，z）的数组，并且数组每一部分是对应着坐标轴的缩放比例。

关于场景绘图很重要的一件事是相对于父结点的变换。如果我们修改了父结点的方向，子结点也会受其影响发生改变。当我们把父结点沿 X 轴移动十个单位，所有的子结点也将会沿 X 轴移动十个单位。最后子结点的方向会根据所有父结点的方向而计算出来。这个概念将会在下面的图标中描述的更加清楚。

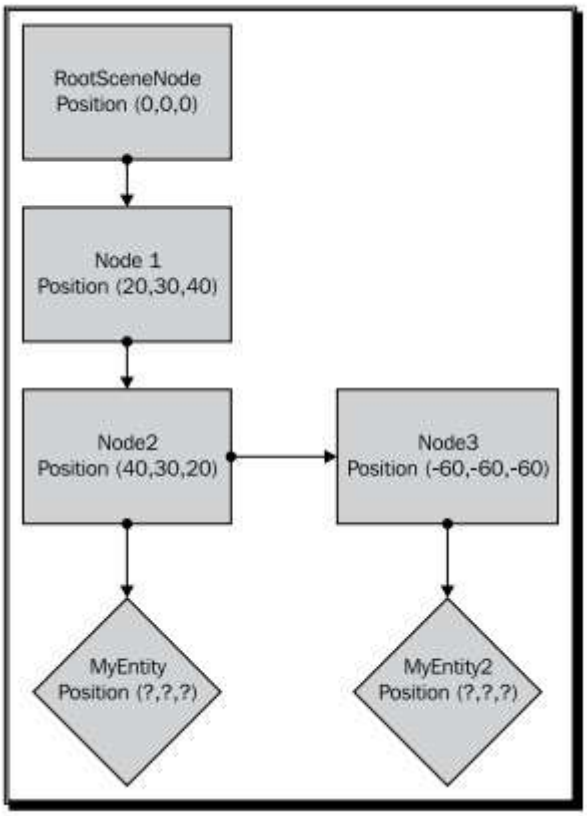


MyEntity 的在场景中的位置将会是(10，0，0)并且 MyEntity2 将会在位置(10，10，20)。然后让我们在 Ogre 3D 中尝试一下这个实验。 】

简单测试——找到场景结点的位置

1. 观察下面的树形结构并且判定 MyEntity 和 MyEntity2 的最终位置

   a. MyEntity(60，60，60) and MyEntity2(0，0，0)
   b. MyEntity(70，50，60) and MyEntity2(10，-10，0)
   c. MyEntity(60，60，60) and MyEntity2(10，10，10)



2.2 设置场景结点的位置

现在，对比上幅图片我们将会尝试创建如图表中描述的场景。

实践时刻 —— 设置场景结点的位置

1. 在创建场景结点后添加以下一行代码：

```
node->setPosition(10，0，0);
```

2. 在 createScene()函数结尾中添加下面一行代码以创建第二个实体。

```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntity2"， "Sinbad.mesh");
```

3. 然后创建第二个场景结点。

```
Ogre::SceneNode* node2 = mSceneMgr->createSceneNode("Node2");
```

4. 把第二个结点添加到第二个结点上

```
node->addChild(node2);
```

5. 设置第二个结点的位置

```
node2->setPosition(0，10，20);
```

6. 把第二个实体关联到第二个结点上面：

```
node2->attachObject(ent2);
```

7. 编译这个程序然后你就会看到两个 *Sinbad* 实例了.

刚刚发生了什么？

我们创建了一个和之前图解中相匹配的场景。我们在第一步使用的首个新函数。能够很容易的猜到，*setPosition(x，y，z)* 函数是根据数组来设置结点位置的。记住这个位置是相对于父结点的位置。我们想要 MyEntity2 在位置(10，10，20)，因为我们添加了关联 MyEntity2 的 node2 结点，并且 node2 的父结点已经在位置(10，0，0)了。我们只需要设置 node2 的位置到(0，10，20)。当两个位置结合到一起，MyEntity2 就在(10，10，20)了。

简单测试 —— 使用场景结点

1. 我们现在有场景结点 node1 在(0，20，0)并且我们有个场景子结点在 node2 并且已经有一个提示关联上去了。如果我们想要实体在(10，10，10)位置被渲染，那么我们应该把 node2 设置在什么位置？

a. (10，10，10)
b. (10，-10，10)
c. (-10，10，-10)

动手试试 —— 添加一个 Sinbad

添加 Sinbad 的第三个实例并且让他在位置(10，10，30)处渲染

2.3 旋转一个场景结点

我们已经知道如何设置一个场景结点的位置。现在，我们将会学习如何去旋转一个场景结点并且如何以另一种方式去修改一个场景结点。 】

实践时刻 —— 旋转一个场景结点

我们将会使用之前的代码，但是会为 createScene()函数写全新的代码。

1. 移走 createScene()函数中的所有代码。

2. 首先创建一个 Sinbad.mesh 的实例并且然后创建一个新的场景结点。设置场景结点的位置到(10，10，0)，在最后关联实体到结点，并且添加一个结点为场景根结点的子结点。

```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity"， "Sinbad.mesh");
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
node->setPosition(10，10，0);
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```

3. 同样的，创建一个新的实例模型，然后是一个新的场景结点，并且设置点到(10，0，0)

```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntity2"， "Sinbad.mesh");
Ogre::SceneNode* node2 = mSceneMgr->createSceneNode("Node2");
node->addChild(node2);
node2->setPosition(10，0，0);
```

4. 现在添加以下两行到旋转模型并且关联实体到场景结点

```
node2->pitch(Ogre::Radian(Ogre::Math::HALF_PI));
node2->attachObject(ent2);
```

5. 同样的，但是这次使用 yaw 函数代替 pitch 函数并且使用 translate 函数代替 setPosition 函数

```
Ogre::Entity* ent3 = mSceneMgr->createEntity("MyEntity3"， "Sinbad.mesh");
Ogre::SceneNode* node3 = mSceneMgr->createSceneNode("Node3");
node->addChild(node3);
node3->translate(20，0，0);
node3->yaw(Ogre::Degree(90.0f));
node3->attachObject(ent3);
```

6. 同样再次代替 yaw()和 pitch()函数使用 roll()函数旋转 】

```
Ogre::Entity* ent4 = mSceneMgr->createEntity("MyEntity4"， "Sinbad.mesh");
Ogre::SceneNode* node4 = mSceneMgr->createSceneNode("Node4");
node->addChild(node4);
node4->setPosition(30，0，0);
```

```
node4->roll(Ogre::Radian(Ogre::Math::HALF_PI));
node4->attachObject(ent4);
```

7. 编译并且运行程序，并且你将会看到下面的截图 】

刚刚发生了什么?

我们几乎重复了 4 遍我们的代码并且总是改变一些小细节。第一次的代码没有什么特别的。它仅是我们之前写过的一样的代码，但是这个实例会作为我们的参照物，用来对比其在他三个实例改变之后发生了什么。在第四步中，我们添加了下面一行代码：函数 *pitch (Ogre::Radian(Ogre::Math::HALF_PI))* 用来绕 X 轴旋转结点。就如我们之前所说的一样，这个函数接受一个弧度单位作为参数并且我们使用 π/2 ，也就是旋转 90 度。

在第五步中，我们代替 *setPosition(x，y，z)* 函数调用了 *translate(x，y，z)* 函数。*setPosition(x，y，z)* 和 *translate(x，y，z)*函数之间的不同 *setPosition()* 函数仅是设置点，没什么可说。*translate()* 以给定的值设置点的位置，但是它是相对于现在的位置变换。如果一个场景结点在位置(10，20，30)并我们调用 *setPosition(30，20，10)*，那个结点在就在世界空间的位置(30，20，10)。另一方面，如果我们调用 *translate(30，20，10)*，结点就会在位置(40，40，40)。这点区别虽小，但是相当重要。如果我们在正确的环境中使用这两个函数他们都是有效的，比如当我们想要设置结点在场景中的位置，我们将会使用 *setPosition(x，y，z)*函数。然而，当我们想要移动一个已经在场景中设置好的结点，我们将会使用 *translate(x，y，z)*.

同样，我们用 *yaw(Ogre::Degree(90.0f))* 代替 *pitch(Ogre::Radian(Ogre::Math::HALF_PI))* 函数。*yaw()* 函数绕 Y 轴旋转场景结点。我们使用 *Ogre::Degree()* 来代替 *Ogre::Radian()*，当然，*pitch* 和 *yaw* 仍然需要使用一个弧度参数。然而，Ogre 3D 提供了一个可以使编译器自动转换角度到弧度的操作的 *Degree()*类。因此，程序员可以随心所欲使用一个弧度或角度单位来旋转场景结点了。不同的类命令确保类的使用是清楚的，这样防止使用混淆或者可能出错的资源。

第六步介绍了三个不同结点旋转函数，其中最后一个函数— *roll()* 函数。这个函数绕 Z 轴旋转场景结点。同样的，我们可以使用 *roll(Ogre::Degree(90.0f))* 来代替 *roll(Ogre::Radian(Ogre::Math::HALF_PI))*。当程序运行显示了可以一没有旋转的模型和三个

已经旋转过了的模型。最左边的模型没有转动，左边模型的右边那个是绕 X 轴转动过的，中间的模型是绕 Y 轴旋转过的，最右边的模型是绕 Z 轴旋转过的。三个实例都显示了不同旋转函数的效果。简而言之，*pitch()* 函数绕 X 轴旋转，*yaw()* 函数绕 Y 轴旋转，*roll()* 函数绕 Z 轴旋转。我们可以使用 *Ogre::Degree(degree)* 或者 *Ogre::Radian(radian)* 其中一个来明确指明我们想要旋转的程度。

简单测试—— 旋转一个场景结点

1. 哪个是旋转结点的三个函数？
   a. pitch， yawn， roll
   b. pitch， yaw， roll
   c. pitching， yaw， roll

动手试试 —— 使用 Ogre::Degree

修改我们之前输入的代码段，替换现有的 Ogre::Radian 为 Ogre::Degree 或反之亦然，旋转将会保持不变

2.4 缩放一个场景结点

我们已经涉及了三个中的两个可以改变绘图场景的场景。现在，是描述最后缩放操作的时刻了。

实践时刻 —— 缩放一个场景结点

再一次，以我们之前用过的程序段作为我们的开始

1. 移走 createScene() 函数中所有代码并插入以下代码段

```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity"， "Sinbad.mesh");
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
node->setPosition(10，10，0);
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```

2. 同样的，创建一个新实体：

```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntity2"， "Sinbad.
```

3. 现在我们使用一个函数创建一个场景结点并且把它添加为一个子结点。然后同样我们照原来所做。

```
Ogre::SceneNode* node2 = node->createChildSceneNode("node2");
node2->setPosition(10，0，0);
node2->attachObject(ent2);
```

4. 现在，在 *setPosition()* 函数之后，调用下面的一行来缩放模型：

```
node2->scale(2.0f，2.0f，2.0f);
```

5. 创建一个新的实体：

```
Ogre::Entity* ent3 = mSceneMgr->createEntity("MyEntity3"，"Sinbad.mesh");
```

6. 现在我们调用第三步中同样的函数，但是添加一个新增的参数

```
Ogre::SceneNode* node3 = node->createChildSceneNode(
                                    "node3"，Ogre::Vector3(20，0，0));
```

7. 在调用完函数之后，插入这一行以缩放模型：】

```
node3->scale(0.2f，0.2f，0.2f);
```

8. 编译程序并运行，然后就就看到下面的图片了：

刚刚发生了什么？

我们创建一个有缩放模型的场景。在第三步之前没有什么特别的发生。然后我们使用了一个新函数，即是—— *node->createChildSceneNode("node2")*。这个函数是场景结点的一个成员函数并且用给定的名字创建新的场景结点，并且当调用函数时，直接添加新结点到指定的父结点。因此，node2 被添加为 node 的子结点。

在第四步中，我们使用了场景结点的 *scale()* 函数。函数使用数组(x，y，z)来表示场景结点是如何缩放的。想，x，y，z 轴都是参数因子，如(0.5，1.0，2.0)表示场景结点应该在 X 轴上缩小一半，在 Y 轴保持不变，在 Z 轴放大一倍。当然，从严格意义上说，场景结点是不能缩放的，它只保存着不被渲染元数据。更严格的说每个渲染的对象将会代替原有结点造成缩放。所以说，结点只是一个关联子结点和渲染对象的容器或者说是参考框架。

在第六步中，我们又使用了 *createChildSceneNode()* 函数，但是这次有更多的参数。在这个函数中的第二个参数接收一个我们常用的数组（x，y，z）。Ogre3D 也有自己调用 (x，y，z) 的类 *Ogre::Vector3*。除了储存数组，这个类提供了实现基础操作的函数。使它们可以使用线代中的三维向量。这个向量描述了当场景结点被创建起来时，结点的变换。*createChildSceneNode()* 函数使用代替了以下代码：

```
Ogre::SceneNode* node2 = mSceneMgr->createSceneNode("Node2");
node->addChild(node2);
```

或者甚至是

```
Ogre::SceneNode* node2 = mSceneMgr->createSceneNode("Node2");
node->addChild(node2);
node2->setPosition(20，0，0);
```

在最后的一段代码可以被替换成

```
Ogre::SceneNode* node2 = node->createChildSceneNode(
                                        "Node2"，Ogre::Vector3(20，0，0));
```

如果我们省略 Vector3 参数，这就成了第一段的代码。这个函数还有很多版本，我们将会稍后展示。如果您有点迫不及待了，请浏览 Ogre3D 的网上文档 http://www.ogre3d.org/docs/api/html/index.html。除了 *scale()* 函数，也有一个 *setScale()* 函数。这两个函数之间的不同就好像 *setPosition()* 和 *translate()* 函数一样。

简单测试 —— 创建一个场景子结点

1. 说出调用 createChildSceneNode() 函数的两种不同方式

2. 如果这行不用 *createChildSceneNode()* 用何代码来代替？

```
Ogre::SceneNode* node2 = node->createChildSceneNode(
                                                "node1"，Ogre::Vector3(10，20，30));
```

这行代码可以用三行代码来代替。第一行是创建场景结点，第二行是变换结点，第三行是把它绑定到 node 结点上。

```
Ogre::SceneNode* node2 = mSceneMgr->createSceneNode("Node2");
node2->translate(Ogre::Vector3(10，20，30));
node->addChild(node2);
```

动手试试 —— 使用 createChildSceneNode() 函数

使用 *createChildSceneNode()* 函数来重构这章你写的所有代码。 】

2.5 用聪明的方式使用场景绘图

在这部分，我们将会学习如何使用场景绘图的一些特性使得场景绘图更加简单。这也将会扩展我们关于场景绘图的认识。

实践时刻 —— 使用场景结点创建树

这次，我们将会使用除 Sinbad 的另一个模型。

1. 移去 createScene() 函数中的所有代码。

2. 用我们之前的方法创建一个 Sinbad。

```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity"， "Sinbad.mesh");
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
node->setPosition(10，10，0);
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```

3. 现在创建一个可以到处跟着 Sinbad 移动的 ninja。(译者注：Sinbad 是天方夜谭中的水手辛巴达，ninja 是日本忍者。)

```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntitysNinja"， "ninja.mesh");
Ogre::SceneNode* node2 = node->createChildSceneNode("node2");
node2->setPosition(10，0，0);
node2->setScale(0.02f，0.02f，0.02f);
node2->attachObject(ent2);
```

4. 编译并且运行应用程序。当你靠近看 Sinbad 的时候，你将会看到一个绿色的 ninja 的在他的左手边。

5. 现在改变位置到(40，10，0)。

```
node->setPosition(40，10，0);
```

6. 把模型绕 X 轴旋转 180 度。

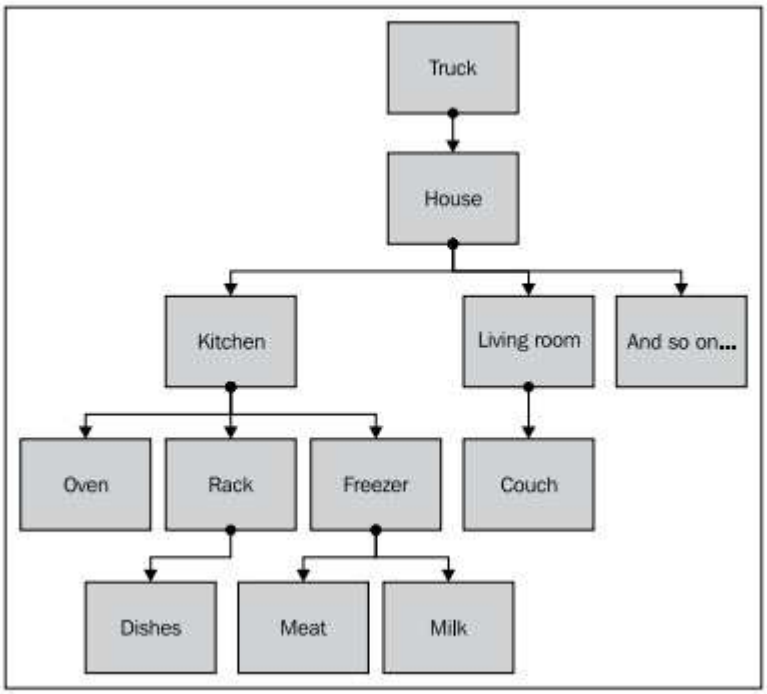
```
node->yaw(Ogre::Degree(180.0f));
```

7. 编译并运行应用程序。

8. 你将会看到 ninja 任然在 Sinbad 的左手边并且 Sinbad 被旋转了

刚刚发生了什么？

我们创建了一个鬼鬼祟祟跟随 Sinbad 的 ninja。我们可以实现是因为我们把 ninja 模型关联为 Sinbad 场景结点的一个子结点。当 Sinbad 移动的时候，我们使用了他的场景结点，所以每步他的变换也会给 ninja，因为他的场景结点是我们设置的 Sinbad 的子结点，并且如我们之前所说，一个父结点的变换将会传递给它所有的子结点。关于场景这一点对创建附加模型和复杂场景极其有用。如是说，如果我们想要创建一个装有房子的卡车，我们会使用多种不同的模型和场景结点。最终，我们将会有一个房子的场景结点和房子内部的东西作为它的子结点。现在我们想要移动房子，我们只需要简单关联房子结点到卡车结点或者别的什么，并且如果卡车移动了，整个房子也将会一起移动。



箭头符号显示出场景绘图方向的变换是沿箭头方向一直传下去的。

简单测试 —— 关于场景绘图

在一个场景绘图中变换是如何传递的？

a. 从叶子结点到根结点。
b. 从根结点到叶子结点。

动手试试 —— 添加一个跟随着的 ninja

添加第二个跟随着第一个日本忍者的忍者到场景之中。

2.6 在场景中的不同空间

在这部分，我们将会学习场景中的不同空间并且如何使用这些空间。

实践时刻 —— 变换世界空间

我们将会以一种与往常不同的方式移动对象。

1. 同样，我们以一个空的 *createScene()* 函数开始；所以在使用前清空函数中的所有代码。

2. 创建一个引用模型。

```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity"， "Sinbad.mesh");
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
node->setPosition(0，0，400);
node->yaw(Ogre::Degree(180.0f));
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```

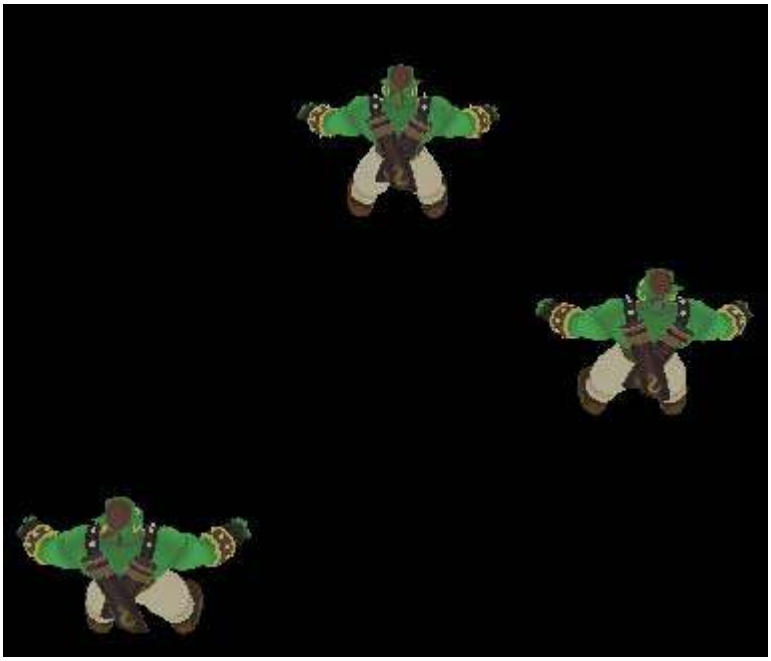
3. 创建两个新的模型实例并且变换每个实例用相对位移(0，0，10)。

```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntity2"， "Sinbad.mesh");
Ogre::SceneNode* node2 = node->createChildSceneNode("node2");
node2->setPosition(10，0，0);
node2->translate(0，0，10);
node2->attachObject(ent2);
Ogre::Entity* ent3 = mSceneMgr->createEntity("MyEntity3"， "Sinbad.mesh");
Ogre::SceneNode* node3 = node->createChildSceneNode("node3");
node3->setPosition(20，0，0);
node3->translate(0，0，10);
node3->attachObject(ent3);
```

4. 编译并且运行应用程序，操纵你的摄像机直到你看到前面的模型得到下面的效果。

5. 把 *node3->translate(0， 0， 10)* 这行代码替换为 *node3->translate(0， 0， 10， Ogre::Node::TS_WORLD);*

6. 同样的，编译并且运行应用程序并且像以前一样操控摄像机。

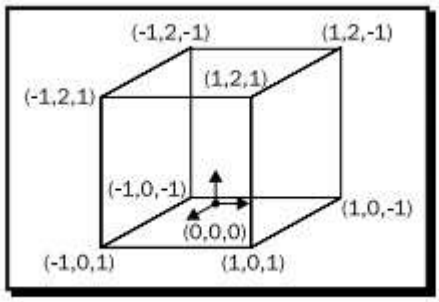


刚刚发生了什么？

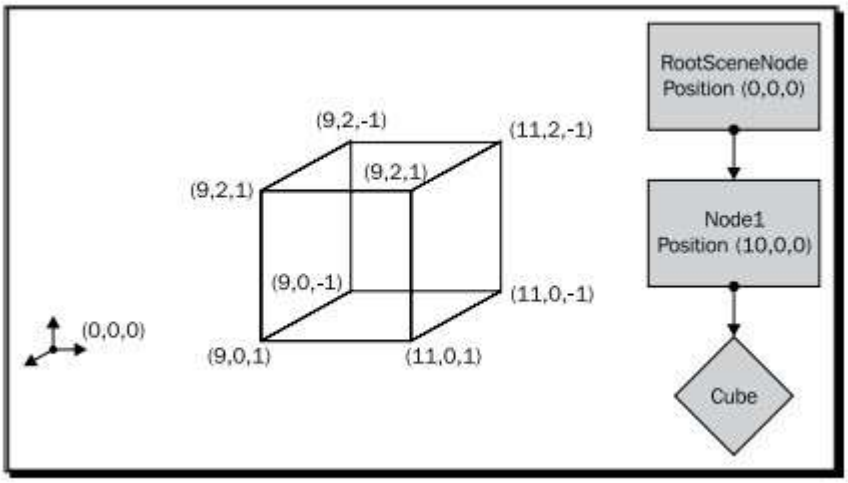
我们使用了 translate() 函数的一个新的参数。这造成了场景中左边的模型相对与中间的模型移动了不同的方向。

在 3D 场景中不同的空间

模型移动方向的不同的因为使用了 *Ogre::Node::TS_WORLD* ，我们告诉 *translate()* 函数变换是发生在世界空间的而非一般的父空间。我们在 3D 场景中有三种空间——世界空间，父空间和局部空间。局部空间是模型本身定义的。如一个立方体有 8 个点并且可以用以下的图来说明：



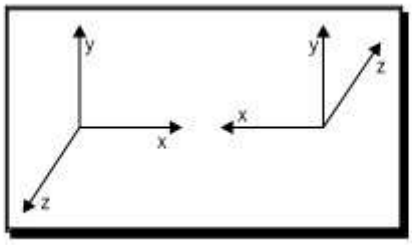
黑色的点是立方体的原点。立方体的每个点都是相对于原点来表示的。当场景被渲染的时候，这个立方体就需要在世界空间中。为了获得在世界空间的立方体的坐标，在场景绘图中立方体中所有结点的应变换适用于世界空间坐标。比方说立方体关联上一个已经关联到场景根结点的结点上面，并且使用变换了(10，0，0)。然后在世界空间中这个立方体就变为了这样：



两个立方体的不同之处在于原点变换了位置，或者更准确的说，这个立方体远离了原点。

当我们调用 *translate()* 函数，如果没有定义使用的空间的话，立方体就相对于父空间移动，就如第五步所做的一样。当没有父结点的立方体被旋转，使用父空间或者局部空间的 *translate()* 函数表现为和世界空间为同一种方式。这是因只有原点的位置改变了并且主轴的

方向并没有改变。当我们说移动立方体(0，0，10)，原点在哪无关紧要—— 只要坐标系的主轴的方向是正确的，变换后的结果不会改变。然而，当一个父结点被旋转后，这个观点就不再正确了。当父结点旋转的时候，父空间的原点的轴也会旋转，同时也会改变 *translate(0，0，10)* 的意义。



左边的坐标系并没有旋转并且（0，0，10）表示移动立方体让观察者感觉到拉近了 10 个单位。这使因为 z 轴是代表着指向屏幕外的方向。当局部空间或父空间的坐标轴旋转 180 度，（0，0，10）就表示立方体远离观察者 10 个单位，因为 Z 轴已经指向了屏幕里面。

我们可以看到达到既定效果，translate() 函数是调用在哪个空间是十分重要的。世界空间主轴的方向总是相同的。更准确的说，世界空间使用了左边的坐标系。父空间的所有旋转使用了上一个父结点自身的坐标系。局部的坐标系包含了全部的旋转，不论是场景结点本身还是所有的父结点。translate()的默认设置是使用父空间。当结点移动使用 translate() 函数，使我们可以旋转场景结点时不需要改变结点的方向。

但是在某些情况下当我们想要在不同的空间发生变换而非在一个父空间。在这些情况下，我们使用 translate()函数的第二个参数。第二个参数定义了我们想要变换发生的空间。在我们代码中，我们使用 Ogre::Node::TS_WORLD 使模型在世界空间中移动，这就变为模型反向旋转，因为我们的模型之前已经旋转过 180 度，这样，X 轴和 Z 轴的方向都改变了。再一次的观察图片查看效果。

2.7 在局部空间中变换

我们已经看到在父空间和时间空间的变换。现在我们将会在局部和父空间变换以区别两者的不同并且对空间之间的不同有个更深的认识。

实践时刻—— 在局部和父空间中变换

1. 再次清空 *createScene()* 函数.

2. 插入一个参考模型。这次我们将会移动模型离我们的摄像器更近，所以我们就不用移动摄像机了。

```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity"， "Sinbad. mesh");
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
node->setPosition(0，0，400);
node->yaw(Ogre::Degree(180.0f));
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```

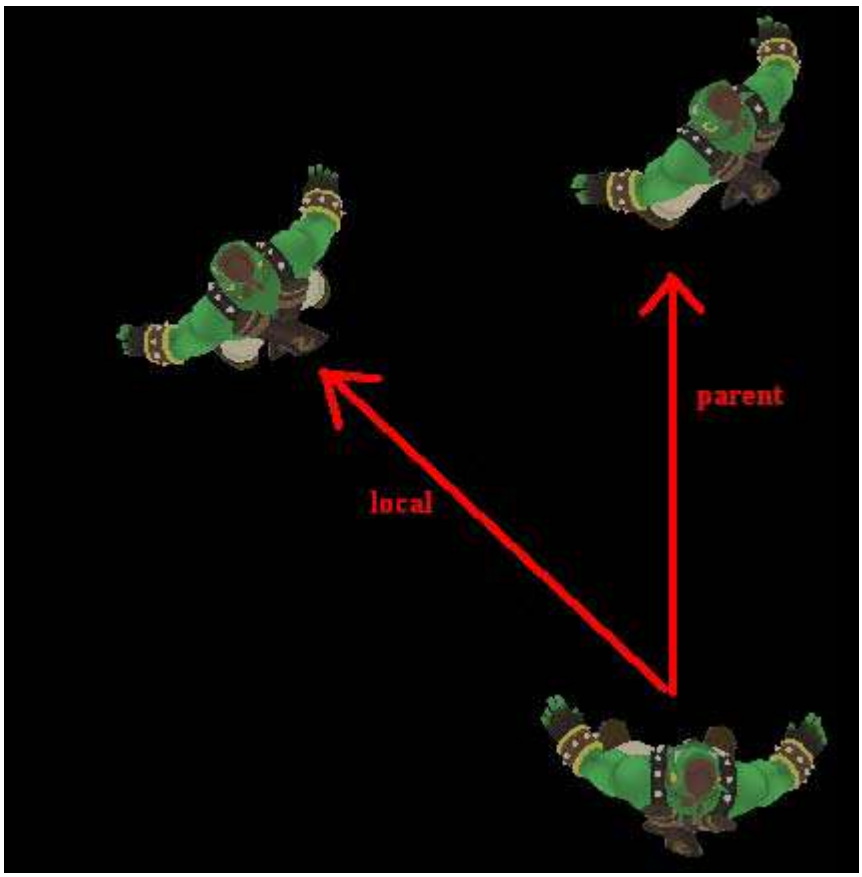
3. 添加第二个模型并且绕 Y 轴旋转 45 度并且在父空间中移动(0，0，20)

```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntity2"， "Sinbad. mesh");
Ogre::SceneNode* node2 = node->createChildSceneNode("node2");
node2->yaw(Ogre::Degree(45));
node2->translate(0，0，20);
node2->attachObject(ent2);
```

4. 添加第三个模型，同样绕 Y 轴旋转 45 度并且在局部空间中移动(0，0，20)

```
Ogre::Entity* ent3 = mSceneMgr->createEntity("MyEntity3"， "Sinbad. mesh");
Ogre::SceneNode* node3 = node->createChildSceneNode("node3");
node3->yaw(Ogre::Degree(45));
node3->translate(0，0，20，Ogre::Node::TS_LOCAL);
node3->attachObject(ent3);
```

5. 编译并运行应用程序。然后再次操控摄像机你就可以从上面看到模型了.



刚刚发生了什么？

我们创建了一个参考模型并且然后添加了两个绕 Y 轴旋转 45 度的模型。然后我们两个都移动(0，0，20)，一个模型在默认的父空间，另一个在局部空间。在父空间的移动过的模型是直接朝 Z 轴移动的。但是因为我们绕 Y 轴旋转了模型，在局部空间中的移动模型，它将会以面向这个角度移动并且最终停在了途中的左上。让我们重复一下。

当我们移动的使用，默认的的设置是父空间，这意味着所有除了旋转过的场景结点都是使用父空间移动的。

当使用世界空间时，旋转是不被考虑进去的。当我们移动的时候，使用的就是就是世界坐标系。

当在局部空间移动的时候，每个旋转，甚至是我们移动过的结点的旋转，都会被用于移动变换。

简单测试——Ogre3D 和空间

指出在 Ogre3D 中不同的 3 个空间

动手试试—— 添加对称

改变 MyEntity2 的旋转和移动使效果图对称。确定你使用了正确的空间。否则，创建一个对称的效果图是十分困难的。这就是之后的效果图。

实践时刻—— 在不同的空间中旋转

这次，我们将会使用不同的空间旋转，如下所述

1. 同样的，我们将会以一个干干净净的 *createScene()* 函数作为开始，所以删除这个函数内的所有代码。

2. 添加参照模型：

```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity"， "sinbad.mesh");
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```

3. 添加第二个模型并且以一般的方式旋转它：】

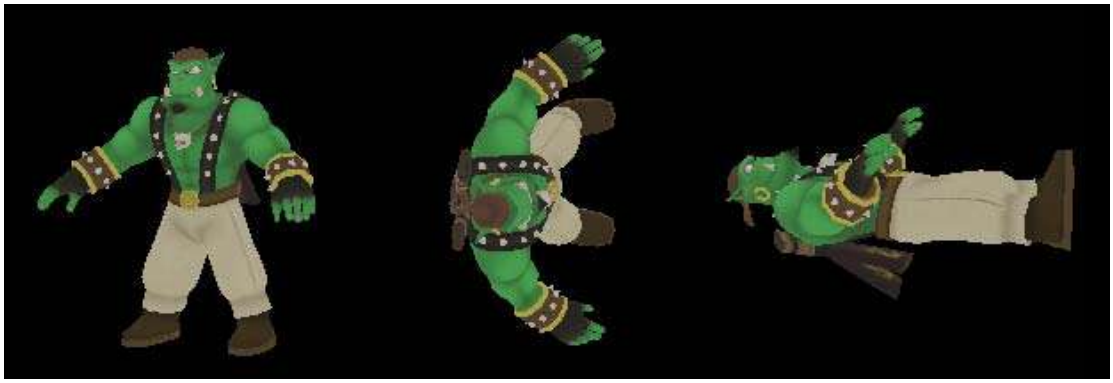
```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntity2"， "sinbad.mesh");
Ogre::SceneNode* node2 = mSceneMgr->getRootSceneNode()->
                                        createChildSceneNode("Node2");
```

```
node2->setPosition(10，0，0);
node2->yaw(Ogre::Degree(90));
node2->roll(Ogre::Degree(90));
node2->attachObject(ent2);
```

4. 使用时间空间添加第三个模型：

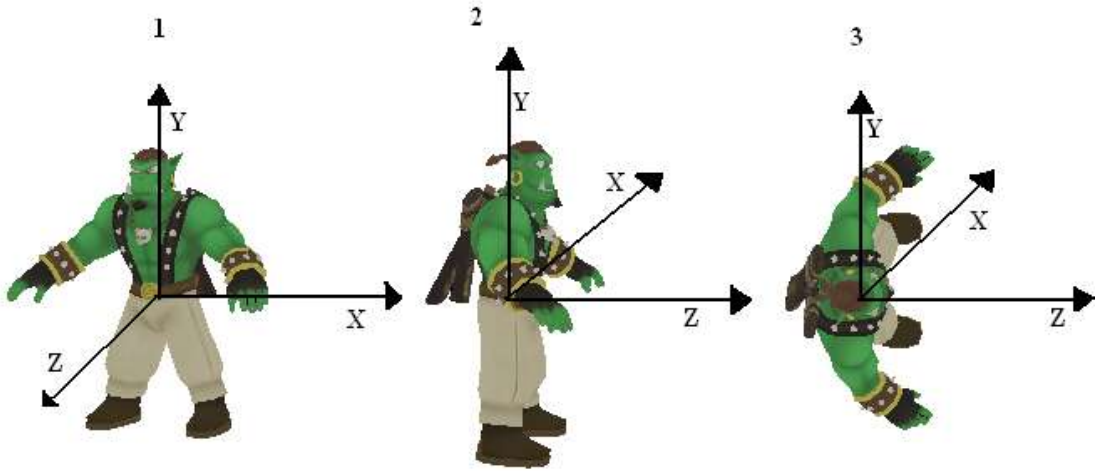
```
Ogre::Entity* ent3 = mSceneMgr->createEntity("MyEntity3"， "Sinbad.mesh");
Ogre::SceneNode* node3 = node->createChildSceneNode("node3");
node3->setPosition(20，0，0);
node3->yaw(Ogre::Degree(90)，Ogre::Node::TS_WORLD);
node3->roll(Ogre::Degree(90)，Ogre::Node::TS_WORLD);
node3->attachObject(ent3);
```

5. 编译并运行程序

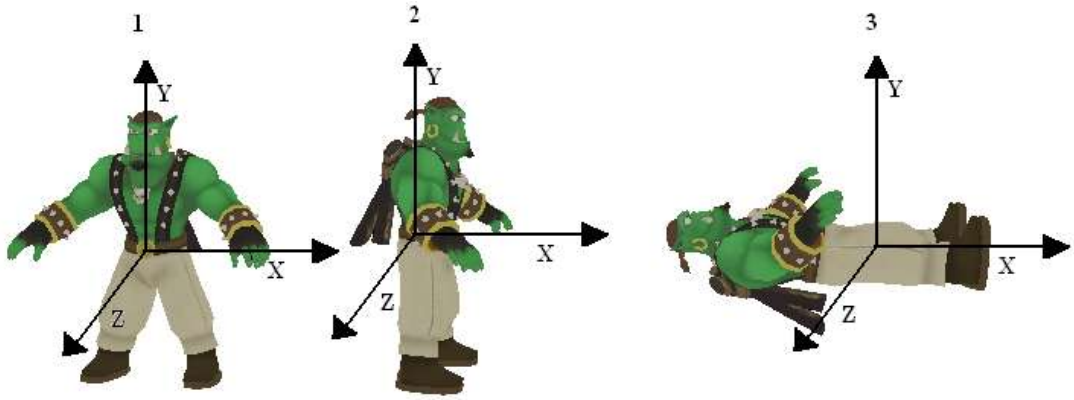


刚刚发生了什么？

如往常一样，我们创建我们的图中左侧的为参考模型。我们旋转第二个模型—— 首先绕 Y 轴旋转并且然后绕 Z 轴旋转。旋转使用默认的空间作为局部的空间。这意味着我们把第一个模型绕 Y 轴旋转 90 度，Z 轴的方向就改变了。第二个模型使用了世界坐标系并且 Z 轴的方向总是保持不变，甚至当我们已经旋转过场景结点。



在一号模型中的坐标系是我们原始的坐标系。在二号中，我们看到经绕 Y 轴旋转 90 度之后的坐标系，在三号中，我们沿 Z 轴旋转 90 度。现在我们将会看代替局部空间使用世界空间后的变换。



虽然我们做同样的旋转，但是因为总是使用世界空间，我们不使用改变了的坐标系，那样会得到一个不同的结果。 】

在不同空间的缩放比例

缩放比例在模型建立之初就完成了，因此在不同空间的缩放是相同的。在每个空间设置缩放比例是没有必要的，因为我们没有必要去做这件事。

2.8 总结

我们在这章学习了很多关于使用 Ogre3D 绘图和用它去创建复杂的场景的知识。

具体有以下几点：

1. 为什么是场景绘图并且它的如何起作用的。

2. 改变位置，方向和结点缩放比例的不同方法。

3. 我们用于旋转和变换的不同坐标系我们如何灵活的使用场景绘图的特性去创建复杂的场景。

之后，在下一章我们将会创建更加复杂的场景，我们将会添加灯光，阴影，并且创建我们自己的摄像机。

第三章 摄像机，光源和阴影

我们已经学习过如何创建一个复杂的场景。但是如果没有光源和阴影，那么这次场景将是不完整的。

在这章，我们将会学习到：

* Ogre3D 支持的不同类型的光源和它们是如何使用的。
* 对一个场景添加阴影和添加可用的不同的阴影技术。
* 什么是摄像机和视口和我们为什么需要使用它们。

3.1 创建一个平面

在我们添加光源到我们的场景之前，我们首先需要添加一个可以投射阴影和光源的平面，这样我们就可以看到阴影了。通常一个应用程序不需要一个平面，因为项目的本身就有可以打上光的地形和地板。光照计算本可以在一个没有平面的程序中，但是那样我们就看不到光源的效果了。

实践时刻——创建一个平面

目前为止，我们总是从一个文件中加载 3D 模型。现在我们就直接创建一个平面：

1. 删除 *createScene()* 函数中的所有代码：

2. 在 *createScene()* 函数中添加下面一行代码来定义一个平面。

```
Ogre::Plane plane(Vector3::UNIT_Y， -10);
```

3. 现在创建一个平面写入到你的内存中。

```
Ogre::MeshManager::getSingleton().createPlane("plane"，
                    ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME，
                    plane， 1500，1500，20，20，true，1，5，5，Vector3::UNIT_Z );
```

4. 创建一个平面的实例。

```
Ogre::Entity* ent = mSceneMgr->createEntity("LightPlaneEntity"， "plane");
```

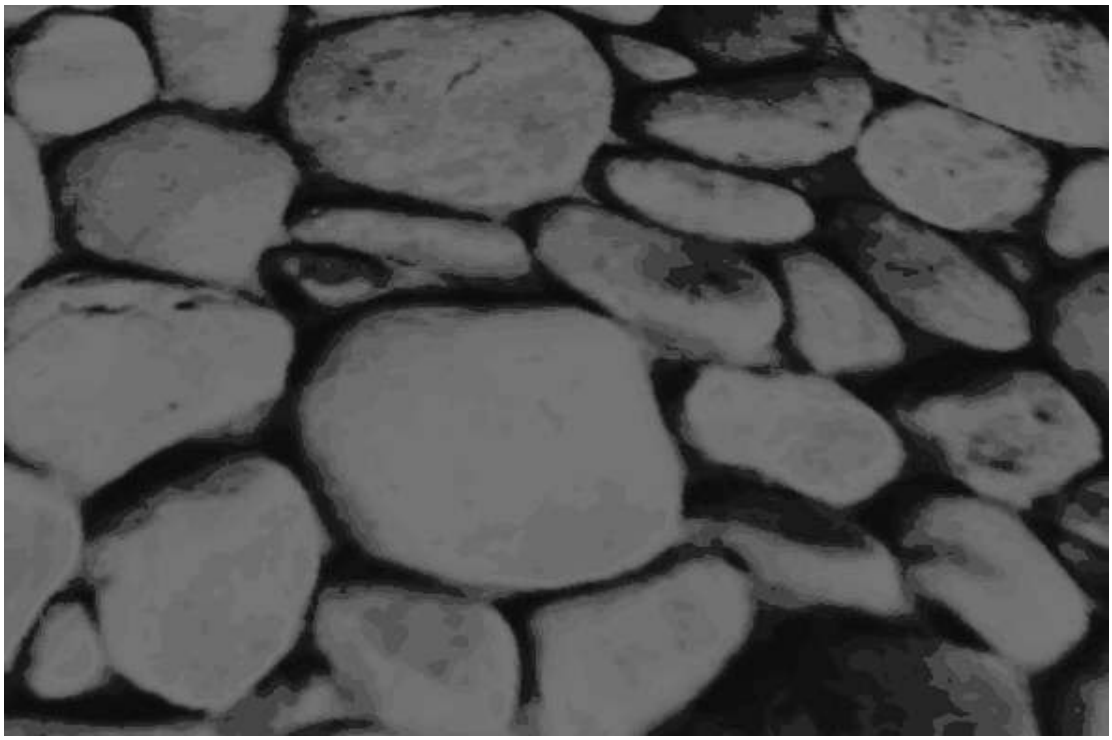
5. 关联平面到场景。

```
mSceneMgr->getRootSceneNode()->createChildSceneNode()->attachObject(ent);
```

6. 为了得到一个不同于白色的平面，设置平面的纹理为一个已经存在的材质。

```
ent->setMaterialName("Examples/BeachStones");
```

7. 编译程序并运行，你将会看到一些暗石头。

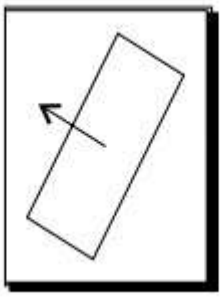


我们转变了文字的颜色以便于阅读！（译者注：指的是纸质书的文字颜色。）

刚刚发生了什么？

我们刚刚创建了一个平面并且把它添加到了场景中。在第二步中我们创建了一个 Ogre::Plane 的实例。这个类描述了一个使用法向量和原点偏移量的平面。

一个法向量(或平面法向量)是在 3D 图形学中一个常用的概念。一个平面法向量指的是一个垂直于平面的向量。法向量的长度通常是 1 并且它被广泛的应用的计算机图形学的光计算和遮挡计算。



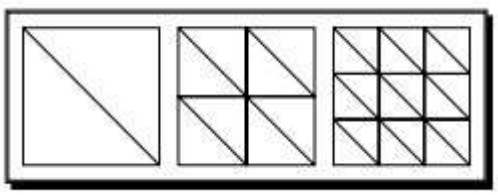
在第三步中，我们使用了一个外面可定义网格的平面。为了实现这个，我们使用了 Ogre MeshManager（Ogre 网格管理器）。这个管理器管理着场景中的网格。除了管理从文件加载的网格，这个管理器也可创建一个由我们自己定义的平面，当然也创建别的一些东西。

```
Ogre::MeshManager::getSingleton().createPlane("plane"，
        ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME，
        plane，1500，1500，20，20，true，1，5，5，Vector3::UNIT_Z );
```

除了定义平面，我们需要给定义的平面一个名称。当从磁盘加载网格的时候，该文件的名称作为该资源的名称。它也需要一个属于它的资源组，这个资源组就好像 C++的命名空间一样。第三个参数是定义的平面然后第四个参数和第五个参数是定义平面的面积大小。第六个和第七个参数是用于描述平面的切片程度。为理解到底什么是切片，我们将会绕个小弯，先给大家讲述一下在 3D 空间中 3D 模型是如何表示的。

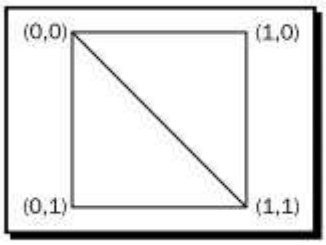
在 3D 空间中表示模型

渲染一个 3D 的模型需要以某种计算机可以理解而且渲染起来很有效率的描述方式。在实时程序中描述 3D 模型的最常见形式就是三角形。我们的平面可以用两个三角形可以组成一个四边形的方式来表示。因为切片的有 X 和 Y 轴的大小的平面参数，我们可以控制用多少三角形来生成一个平面。在下面的图片中，我们将会看到用每个轴一个，二个或者三个三角形切片组成平面。为看到这种效果，我们运行程序然后按下 R 键。这样就可以从第一渲染模式变为线框模式，这样我们就可以看到三角形了。再按一下 R 键将会改变现有模式为点模式，我们将会三角形的顶点了。再按一下 R 键将会改变为正常的渲染模式。



在定义完我们想要的切片的程度，我们传递一个布尔的参数来告诉 Ogre 3D 平面的法

向量是否被计算。正如之前所述，法向量是垂直于平面的一个向量。最后的那三个参数是作用用于纹理。渲染的纹理时，所有的点都需要纹理坐标。纹理坐标告诉渲染引擎如何映射材质到三角形。因为一张图片是一个 2D 的表面，纹理的坐标包含两个值即—— x 和 y 。它们被表示为一个二元组(x，y)。纹理坐标值正常初始化的范围为从 0 到 1。(0，0)表示纹理的左上角，（1，1）表示为右下角。有时候它们的值会大于 1，这表示纹理可以根据设置模式来进行重复。这个话题我们将在接下来的章节展开来谈。(2，2)可能表示纹理横跨两轴重复两次。第十和第十一个参数告诉 Ogre 3D 我们想要纹理平铺平面的频率。第九个参数定义了我们需要多少个纹理坐标系。当我们使用超过 1 个的表面纹理的时候，这个参数就变的很有用了。最后一个参数定义了纹理”up”的方向。这也会影响到纹理坐标的生成。我们简单的说 Z 轴应”up”我们的平面。



在第四步，我们创建了一个刚经过 MeshManage(网格管理器)创建的平面的实例。要做到这一点，我们需要使用在创建过程中我们给予平面的名称。在第五步中我们关联实体到场景。

在第六步中，我们设置了实体实例的一个新材质。每个实体都会有一个材质分配给它。这个材质描述了我们使用的纹理与其所具有的光效果与材质的相互作用。在我们设置这个创建好的平面纹理之前，它将会被渲染为白色。因为我们想要看到创建的光源的效果，但是白色不是可使用的最佳颜色。我们使用了一个已经在 media 文件夹下定义的材质。这个材质方便的对平面的添加了石头纹理。

3.2 添加一个点光源

现在我们已经创建了一个可以在我们场景中看见光源效果的平面，我们需要添加一个光源去看下效果。

实践时刻 —— 添加一个点光源

我们将会创建一个点光源并且添加到我们的场景中，然后观察光源在我们场景中的效果:

1. 在设置完平面的材质之后添加以下代码：

```
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
mSceneMgr->getRootSceneNode()->addChild(node);
```

2. 创建一个名为 Light1 的光源并且告诉 Ogre3D 这是一个点光源：

```
Ogre::Light* light1 = mSceneMgr->createLight("Light1");
light1->setType(Ogre::Light::LT_POINT);
```

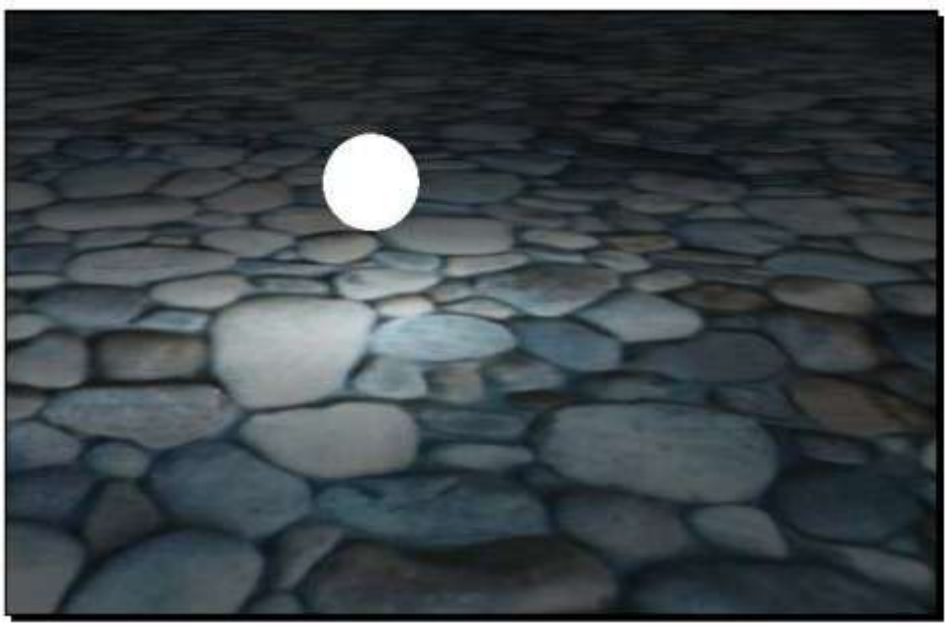
3. 设置光源的颜色和位置：

```
light1->setPosition(0，20，0);
light1->setDiffuseColour(1.0f，1.0f，1.0f);
```

4. 创建一个球并且设置它在点光源的位置，这样我们就可以看到光源的位置了。

```
Ogre::Entity* LightEnt = mSceneMgr->createEntity("MyEntity"， "sphere.mesh");
Ogre::SceneNode* node3 = node->createChildSceneNode("node3");
node3->setScale(0.1f，0.1f，0.1f);
node3->setPosition(0，20，0);
node3->attachObject(LightEnt);
```

5. 编译运行程序，你应会看到石头的纹理将会被一个白色的光源照亮，并且会看到在平面上面有一个白色的圆球。



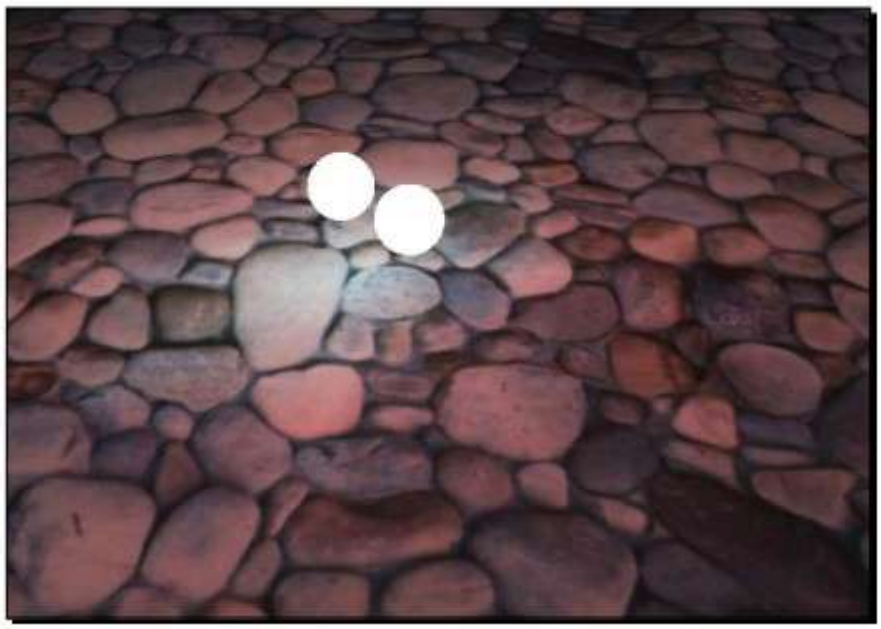
刚刚发生了什么？

我们在场景中添加了一个点光源并且使用了一个白色的球体来标明点光源的位置。在第一步中，我们创建了一个场景结点并添加它到我们的场景根结点。我们创建场景结点是因为我们要为稍后关联白色球体做准备。第一个比较有趣的事情是发生在第二步。我们使用场景管理器创建了一个新光源。如果我们给光源一名称，每个光源的名字必须是独一无二的，如果我们不给光源名称，然后 Ogre 3D 会为我们生成一个。

我们使用 Light1 作为光源的名称。创建之后，我们告诉 Ogre 3D 我们想要创建一个点光源。我们可以创建三种不同的光源，即为——点光源，聚光灯和方向光源。在这我们创建了一个点光源。一会我们将会创建别的类型的光源。一个点光源可以被认为是一个明亮的灯泡。它就好像是空间中可以照亮周围一切的光源。在第三步中，我们使用了刚创建的光源并且设置了光源的位置和颜色。每个光源的颜色是用一个(r，g，b)的数组来描述的。所有三个参数的范围都是从 0.0 到 1.0 而且每个参数表述了它们各自对应颜色属性对最终颜色效果的影响。’r’代表红色，’g’代表绿色，’b’代表蓝色。（1.0，1.0，1.0）是白色，(1.0，0.0，0.0)为红色，其他等等如此类推。我们调用的函数 setDiffuseColour(r，g，b)中的三个参数恰恰是对应表述颜色的(r，g，b)三个参数。在第四步在光源的位置添加了一个白色球体，这样我们就可以看到光源在场景中的位置了。

动手试试 —— 添加第二个点光源

在(20，20，20)的位置添加第二个照亮场景的红色点光源。同样的添加另一个球体以显示点光源的位置。以下就是效果图:】



3.3 添加一个聚光灯

我们已经创建了一个点光源而现在我们将会创建一个聚光灯——第二种我们可以使用的光源类型。

实践时刻 —— 创建一个聚光灯

我们将会使用我们之前已经写好的代码并且简单修改一下，然后观察一个聚光灯是如何工作的：

1. 删除我们创建光源的代码并插入以下代码以创建一个新的场景结点。注意不要删除我们使用过的 LigthEnt 的代码段，然后添加以下代码：

```
Ogre::SceneNode* node2 = node->createChildSceneNode("node2");
node2->setPosition(0，100，0);
```

2. 同样的，创建一个光源，但是现在设置光源的类型为 spotlight

```
Ogre::Light* light = mSceneMgr->createLight("Light1");
light->setType(Ogre::Light::LT_SPOTLIGHT);
```

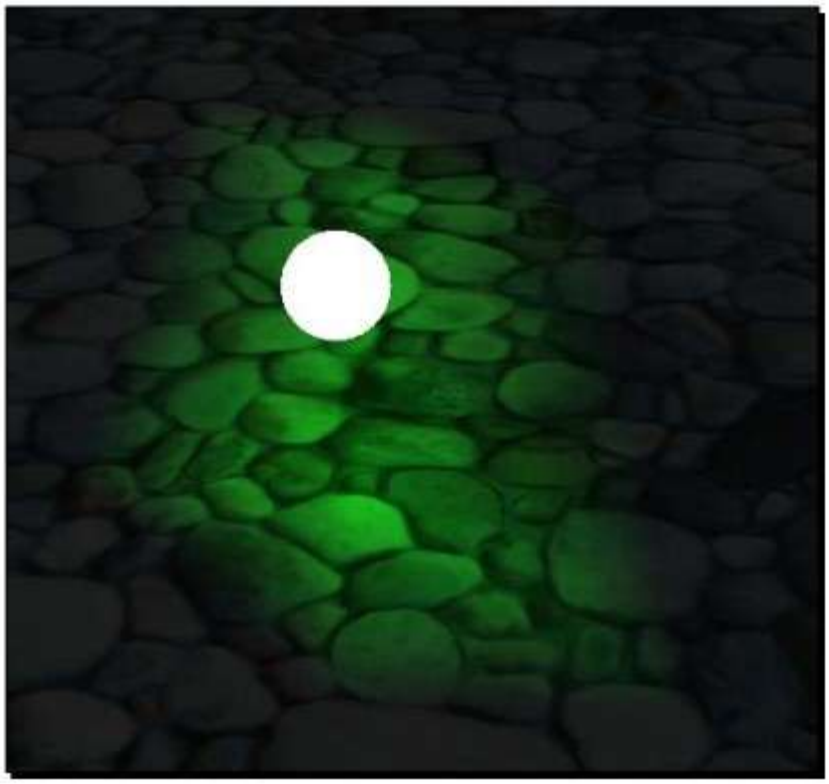
3. 现在设置一些参数，我们将会稍后讨论他们的意思。

```
light->setDirection(Ogre::Vector3(1，-1，0));
light->setSpotlightInnerAngle(Ogre::Degree(5.0f));
light->setSpotlightOuterAngle(Ogre::Degree(45.0f));
light->setSpotlightFalloff(0.0f);
```

4. 设置光源的颜色，然后添加光源到刚创建的场景结点：

```
light->setDiffuseColour(Ogre::ColourValue(0.0f，1.0f，0.0f));
node2->attachObject(light);
```

5. 编译运行程序。它将会有以下效果：

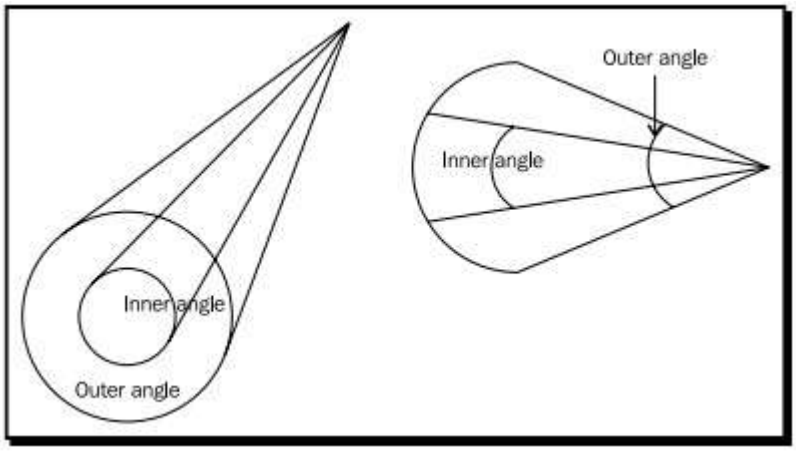


刚刚发生了什么？

我们几乎以创建点光源同样的方式创建了一个聚光灯。不同的是我们改变了光源的一些参数。在第一步中，我们创建了在稍后会用到的场景结点。在第二步中，我们如往常一样创建了一个光源，但是我们使用了不同的光源类型——这次我们使用了 Ogre::Light::LT_SPOTLIGHT —— 来获取一个聚光灯。在第三步是有趣的的步，我们为聚光灯设置了不同的参数。

聚光灯

聚光灯恰如手电筒的效果。它们有发光源的位置和在照亮场景的一个方向。设置光线方向是我们创建聚光灯完成后要做的第一件事。光线的方向简单的定义了聚光灯的指向。接下来的我们设置的两个参数为聚光灯的内角度和外角度。聚光灯的内光部分使用完整的光源颜色来照射区域，而外部的锥体只使用较少的光源能量来照亮物体。这样做是为了模拟手电筒的效果。一个真正的手电筒也是有内光部分和外光部分，其中外光部分没有聚光灯的中央光源的亮度强。我们的定义的内角和外角决定了光源照射的内部和外部的范围有多大。在设置完角度之后，我们设置了一个 下降(falloff) 参数.这个下降参数描述了当照射外层锥体光能损失。被照射的点离内部的距离越大，下降效果就越明显。如果一个点是在圆锥体之外，那它将不会被聚光灯所照射到。



我们设置下降为 0。理论上，我们应该在平面上看到一个完美的光圈，但是实际效果却发生很大的模糊和变形。造成这种效果的原因是我们此刻使用的是平面上的三角形的点去计算光照并应用于照射。当创建一个平面时，我们告诉 Ogre 3D 以 20*20 的切片程度来创建平面。如此大的平面却用如此低的分辨率，这就意味着光线不能被准确的计算，因为在区域内只有很少的点能适用于形成一个边缘光滑的圆形。因此，为了获得一个更好的渲染效果，我们不得不增加平面的切片数。比方说我们增加切片从 20 到 200。那么平面创建代码在切片增加过后就如下面的形式：

```
Ogre::MeshManager::getSingleton().createPlane("plane"，
        ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME，
        plane，1500，1500，200，200，true，1，5，5，Vector3::UNIT_Z );
```

现在当我们重新编译运行程序，我们将会从我们聚光灯得到一个更圆的光圈。

这个圆任然不是完美的。如果需要，我们可以增加平面的切片程度，也可以把光源放远一点使它看起来更加完美。也有不同的光源技术使低分辨率的平面达到更好的效果，但是它们却相当复杂并且使事情之间变得复杂。但是即使是复杂的光源技术，其基本的原理是相同，或者是通过改变创建光源的光源策略。

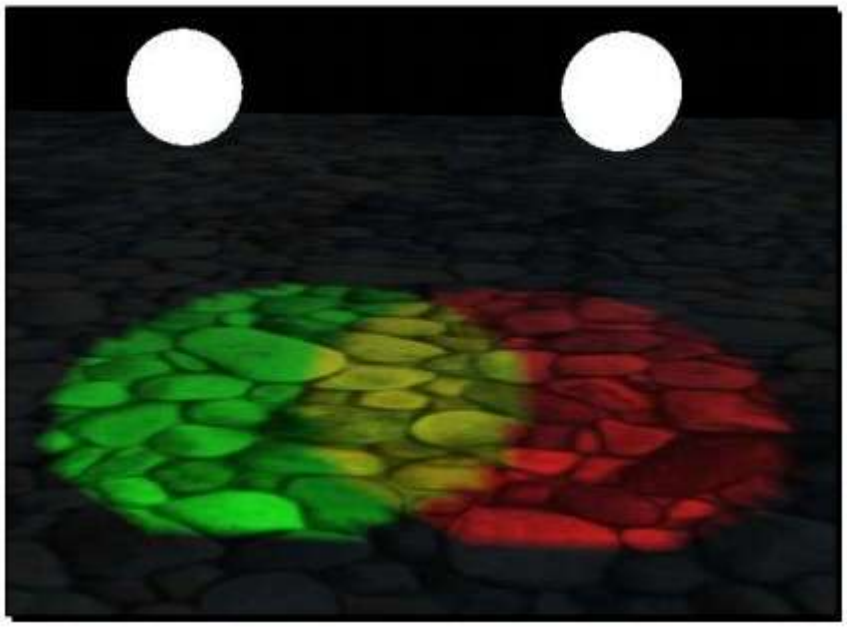
在第四步中，我们看到了在 Ogre3D 中描述颜色的另一种方式。在此之前，我们使用三个值(r，g，b)， 来设置我们光源的漫射色。这里我们使用了 Ogre::ColourValue (r，g，b) ，两种表述方式基本一样，但是这种方式加以一些额外的函数作为一个类被封装，从而使得参数的意图更为清晰。

简单测试 —— 不同的光源资源

用几句话描述点光源和聚光灯之间的不同。

动手试试 —— 混合光线颜色

创建与第一个聚光灯位置不同的第二个聚光灯，给第二个聚光灯以红色的光线，以如此方式安置光源，可以使两个聚光灯重叠一部分。你就可以看到在绿色和红色的重叠区域会有颜色的混合。



3.4 方向光源

我们已经创建过了聚光灯和点光源。现在我们准备创建最后一种光源类型—— 方向光源。方向光源是一种离你很远的光源而且这种光只有方向和颜色，但是却没有像聚光灯和点光源的锥形光束和光照范围。它可以被认为是太阳光。对于我们而言，阳光是从一个方向照射过来的，这个方向也就是阳光的方向。

实践时刻 —— 创建一个方向光源

1. 除了与平面相关的代码，删除所有的 *createScene()* 函数中的旧代码。】

2. 创建一个光源并且设置光源的类型为方向光源：

```
Ogre::Light* light = mSceneMgr->createLight("Light1");
light->setType(Ogre::Light::LT_DIRECTIONAL);
```

3. 设置光源为白色并且设置光源的方向为右下方。

```
light->setDiffuseColour(Ogre::ColourValue(1.0f，1.0f，1.0f));
light->setDirection(Ogre::Vector3(1，-1，0));
```

4. 编译运行程序。

刚刚发生了什么？

我们创建了一个方向光源并且使用 *setDirection(1，-1，0)* 设置它发光的方向是右下。在之前的例子中，我们创建的平面几乎是黑色并且平面只有一小部分被点光源或聚光灯所照亮。这里，我们使用了一个方向光源，这以后整个平面被照亮了。正如前面所述，方向光源可以认为是一个太阳，太阳发出的光是没有衰减半径，也没有别的特殊性质。所以当太阳发光的时候，它会照亮所有的物体。这对我们的方向光源同样适用。

简单测试 —— 不同的光源类型

回忆 Ogre 3D 中的三种光源类型并且说明之间的不同

遗漏的东西

我们已经添加光源到我们的场景中，但是却遗漏了一些东西。在下个例子中我们将会显示出到底是遗漏了什么。

实践时刻 —— 找出到底遗漏了什么

我们使用之前推荐的代码来找出在场景中遗漏了什么东西。

1. 在创建光源完成后，添加代码以创建一个 Sinbad.mesh 的实例并创建一个节点用以关联模型。

```
Ogre::Entity* Sinbad = mSceneMgr->createEntity("Sinbad"， "Sinbad.mesh");
Ogre::SceneNode* SinbadNode = node->createChildSceneNode("SinbadNode");
```

2. 然后按三倍的大小来设置 Sinabad 的缩放比例，并把它稍往上移动一点。否则，它将卡在平面上。同样的，添加它到场景，这样它就可以被渲染到了。

```
SinbadNode->setScale(3.0f，3.0f，3.0f);
SinbadNode->setPosition(Ogre::Vector3(0.0f，4.0f，0.0f));
SinbadNode->attachObject(Sinbad);
```

3. 编译运行程序。

刚刚发生了什么？

我们在场景中添加了一个 Sinbad 的实例。我们的场景仍然是发亮的，但是我们看到 Sinbad 却不投射出影子，这就相当的不切实际了。下一步就是添加阴影到我们的场景。

3.5 添加阴影

一个没有阴影的 3D 场景不是真正完整的 3D 场景。因此，让我们添加它们

实践时刻 —— 添加阴影

使用之前已使用过的代码

1. 在 createScene() 函数中现存的代码之后添加以下一行：

```
mSceneMgr->setShadowTechnique(Ogre:: SHADOWTYPE_STENCIL_ADDITIVE);
```

2. 编译运行程序。

刚刚发生了什么？

用了刚才的一行代码，我们添加阴影到我们的场景之中。Ogre 3D 为我们做了在剩下的工作。Ogre 3D 支持不同的阴影技术。我们使用了 additive stencil shadows。Stencil 意思是指，当渲染场景时，使用的一种特殊的纹理缓冲。

Additive 意味着场景在画面视角中渲染一次并且每个光源的效果积累为最终的渲染效果。这种技术产生了很好的效果，但是却付出了昂贵的代价。因为每添加一个光源渲染的运转就会增加。我们不能深入细节讨论这种阴影的工作原理是什么，因为这实在是一个很复杂的领域。有很多关于这个专题的书籍，而且，阴影的技术在快速改变并且它被人们大量的研究。如果你对这个专题很感兴趣。你可以寻找关于 NVIDIA 的有关 GPU 的宝石系列丛书或 ShaderX 系列丛书中有意义的文章或者寻找 Siggraph（计算机图形图像特别兴趣小组）的会议记录（http://www.siggraph.org/）。

3.6 创建摄像机

目前为止，我们总是使用 *ExampleApplication* 类中创建的摄像机。现在让我们自己创建一个摄像机。摄像机，顾名思义，从一个确切的位置来捕捉我们的一部分场景。在某一特定时间只有一个活动的摄像机，那是因为我们仅有一个输出媒体，那就是我们的显示器。但是在场景中也有可能使用数个摄像机当每个摄像机陆续的被渲染。

实践时刻 —— 创建一个摄像机

这次我们不修改 *createScene()* 函数；所以保留 Sinbad 的实例和阴影。

1. 在 ExampleApplication 中创建一个新的名为 createCamera()的空函数

```
void createCamera() {
}
```

2. 创建一个新的称为 MyCamera1 的摄像机并把它分配给数据成员 mCamera：

```
mCamera = mSceneMgr->createCamera("MyCamera1");
```

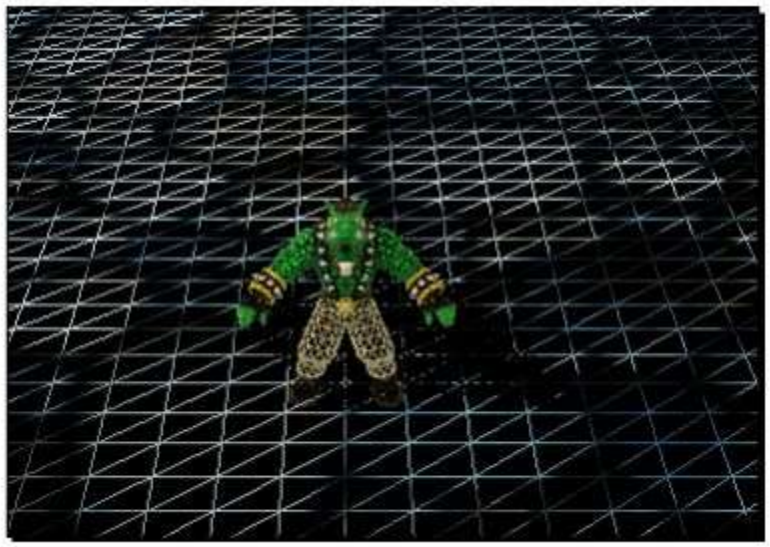
3. 设置摄像机的位置并让其镜头朝向原点：

```
mCamera->setPosition(0，100，200);
mCamera->lookAt(0，0，0);
mCamera->setNearClipDistance(5);
```

4. 现在改变渲染模式至线框模式 】

```
mCamera->setPolygonMode(Ogre::PM_WIREFRAME);
```

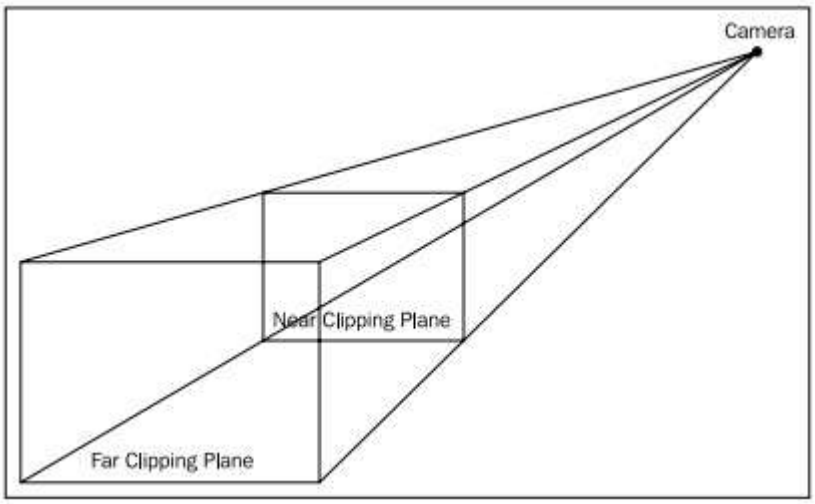
5. 编译运行程序。



刚刚发生了什么？

我们重载了最初创建摄像机的 createCamera() 函数并设置它到一个位置。在创建之后，我们设置完它的位置并且使用 lookat()函数以设置摄像机的镜头对准原点。我们所做的下一步是设置剪裁的距离。

一个摄像机只可以看到部分的 3D 场景。因为完整渲染需要浪费宝贵的 CPU 和 GPU 时间。为避免这种情况，在渲染之前，场景管理器(SceneManager)将会把大部分的场景从场景中裁剪出去。只有摄像机的可见部分被渲染到。这一步叫做拣选。只有位于远近裁剪面并且在视锥体内部的物体才会被渲染。这被称为摄像机的视锥。视锥没有顶部的锥体。只有在剪裁过的锥体内部的对象才能被摄像机所看到。更多信息可在 http://www.lighthouse3d.com/opengl/viewfrustum/ 中找到



然后我们改变渲染模式为线框模式。

在重载 *createCamera()* 函数之前，摄像机的起始位置是悬停在平面上方一点，镜头朝向原点。使用 *setPosition(0，100，200)*，设置我们的摄像机到更高的位置。下面的截图显示了改变的效果。

动手试试 —— 做更多的事

尝试设置摄像机到不同位置和使用摄像机不同镜头朝向，查看摄像机在初始位置发生的效果。 同样也尝试一下增加近距离剪裁并试验这种效果是什么。下图可能是我们看到的效果，我们可以看到 Sinbad 的头部的内部(译注：图中红色的 Sinbad 的大舌头^_^)。近距离剪裁设置到 50 可以产生这种效果。

3.7 创建一个视口

与摄像机概念紧密结合的一个概念就是视口。所以我们也将会创建我们自己的视口。视口是一个被用来渲染的 2D 表面。我们可以把它当做呈现照片的底片。这种底片有一个底色而且如果图像没有覆盖这个区域，那么我们将会看见底色。

实践时刻 —— 用“行动”举例说明

我们将会使用之前的代码并且在此创建一个新的方法：

1. 删除在 *createCamera()* 函数中调用的 *setShadowTechnique()* 函数。

2. 创建一个空的 *createViewports()* 方法：

```
void createViewports() {
}
```

3. 创建一个视口：

```
Ogre::Viewport* vp = mWindow->addViewport(mCamera);
```

4. 设置背景颜色和纵横比：

```
vp->setBackgroundColour(ColourValue(0.0f，0.0f，1.0f));
mCamera->setAspectRatio(Real(vp->getActualWidth()) / Real(vp->getActualHeight()));
```

5. 编译运行程序。

刚刚发生了什么？

我们创建一个视口。创建的时，我们需要传递一个摄像机作为函数的参数。每个视口只可渲染一个摄像机的视野，所以在传参时，Ogre 3D 强制函数只能接受一个摄像机作为参数。当然，摄像机可适当地使用 getter 和 setter 函数以发生改变。最值得注意的改变是背景色从黑色变为蓝色。原因很明显：在第三步中，新的视口设置背景色为蓝色。同样在第三步，我们设置了纵横比——纵横比描述了当渲染图像时宽和高的比例。数学公式为：纵横比 = 窗口宽度 / 窗口高度

动手试试 —— 使用不同的纵横比

尝试使用不同的纵横比并查看图形产生的不同效果。同样的，改变背景色并查看效果。下面的图片是纵横比的宽度设置为 1/5 的效果图。

3.8 总结

在这一章，我们添加光源和阴影到我们的场景，创建了视口并且了解了无锥顶的视锥体。

具体来说，我们讨论：

* 什么是光源和他们如何修改场景的外观。
* 添加阴影到我们的场景
* 创建我们自己的摄像机，视锥体和视口

在下一章，我们将会学习如何处理用户的键盘和鼠标输入。我们将会学习什么是帧监听并了解如何使用它。

第四章 获取用户输入和使用帧监听

迄今为止，我们总是创建静止的场景并且在场景中没有移动的物体。在这一章，我们将会改变这种现状。

在这章，我们将会：

* 认识什么是帧监听
* 认识如何处理用户输入
* 结合两种概念创建我们自己的摄像机控制

4.1 准备一个场景

在添加移动物体之前，我们首先应创建一个可添加物体的场景。然后让我们一起创建一个场景吧。

实践时刻 —— 准备一个场景

我们将使用和之前章节中略微不同的创建场景的版本。

1. 删除 *createScene()* 和 *createCamera()* 函数中的所有代码：

2. 删除 *createViewports()* 函数。

3. 添加一个新的成员变量到类中。这个成员变量是一个场景结点的指针：

```
private:
   Ogre::SceneNode* _SinbadNode;
```

4. 使用 *createScene()* 函数创建一个平面并添加它到场景之中：

```
Ogre::Plane plane(Vector3::UNIT_Y，  -10);
Ogre::MeshManager::getSingleton().createPlane("plane"，
                ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME，
                plane，1500，1500，200，200，true，1，5，5，Vector3::UNIT_Z );
Ogre::Entity* ent = mSceneMgr->createEntity("LightPlaneEntity"，  "plane");
mSceneMgr->getRootSceneNode()->createChildSceneNode()->attachObject(ent);
```

```
ent->setMaterialName("Examples/BeachStones");
```

5. 然后添加一个光源到场景中：

```
Ogre::Light* light = mSceneMgr->createLight("Light1");
light->setType(Ogre::Light::LT_DIRECTIONAL);
light->setDirection(Ogre::Vector3(1，-1，0));
```

6. 我们也需要一个 Sinbad 的实例，创建一个结点并关联实例。

```
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
mSceneMgr->getRootSceneNode()->addChild(node);
Ogre::Entity* Sinbad = mSceneMgr->createEntity("Sinbad"， "Sinbad.mesh");
_SinbadNode = node->createChildSceneNode("SinbadNode");
_SinbadNode->setScale(3.0f，3.0f，3.0f);
_SinbadNode->setPosition(Ogre::Vector3(0.0f，4.0f，0.0f));
_SinbadNode->attachObject(Sinbad);
```

7. 我们也想在场景中增加阴影；所以添加它们：

```
mSceneMgr->setShadowTechnique(SHADOWTYPE_STENCIL_ADDITIVE);
```

8. 创建一个摄像机并放置它在(0，100，200) 并把镜头朝向（0，0，0）；记住添加代码到 *createCamera()* 函数中。

```
mCamera = mSceneMgr->createCamera("MyCamera1");
mCamera->setPosition(0，100，200);
mCamera->lookAt(0，0，0);
mCamera->setNearClipDistance(5);
```

9. 编译运行，你将会看到下面图中的效果

刚刚发生了什么？

我们使用在之前章节学到的东西创建了一个场景。我们应可以理解发生了什么。如果没有的话，我们就应该回过头再看看之前的章节直到我们可以理解为止。

4.2 添加运动到场景之中

我们已经创建了场景；现在让我们把运动添加进场景中。

实践时刻 —— 添加运动到场景中

目前为止，我们只有一个类，即是，*ExampleApplication*。这次我们需要另一个类：

1. 创建一个新类，命名为 *Example25FrameListener*，并使它公有继承 *Ogre::FrameListener*

```
class Example25FrameListener : public Ogre::FrameListener
{};
```

2. 添加一个私有的成员变量，一个 Ogre::SceneNode 的指针，并把它命名为_node

```
private:
   Ogre::SceneNode* _node;
```

3. 添加接收一个 *Ogre::SceneNode* 指针为参数的公有构造函数，并把这个指针赋值操作给成员变量 node 指针。

```
public:
   Example25FrameListener(Ogre::SceneNode* node)
   {
    _node = node;
   }
```

4. 添加一个新的 *frameStarted(FrameEvent& evt)* 函数，在函数中用(0，0，0.1)变换成员变量 node，然后返回 true：

```
bool frameStarted(const Ogre::FrameEvent    &evt)
{
   _node->translate(Ogre::Vector3(0.1，0，0));
   return true;
}
```

5. 添加一个 FrameListener 指针为成员变量。

```
Ogre::FrameListener* FrameListener;
```

6. 添加一个初始化 FrameListener 为 NULL 的构造函数和一个当程序结束时删除 FrameListener 的析构函数。

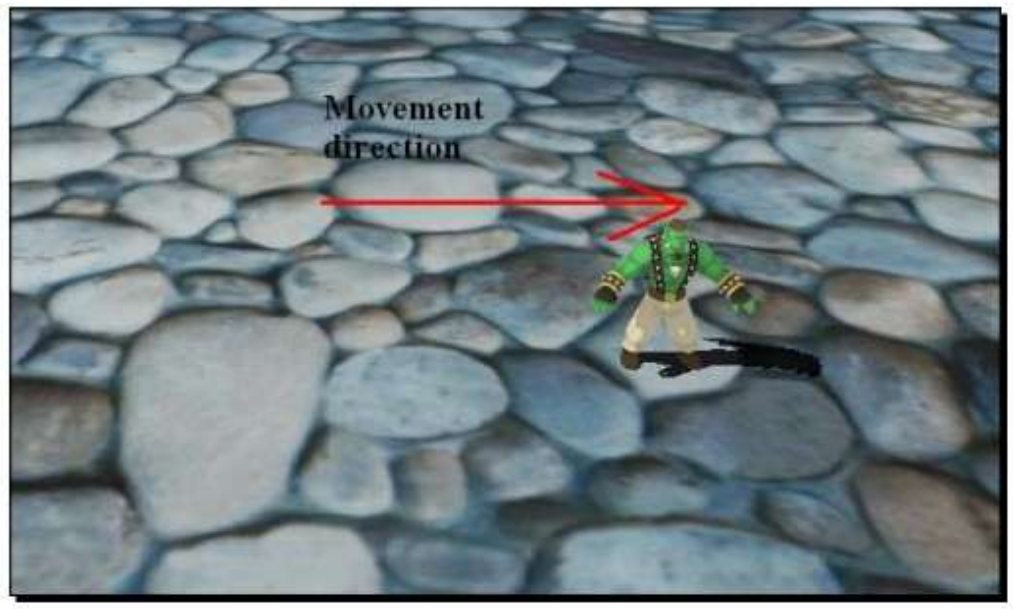
```
Example25()
{
   FrameListener = NULL;
}
~Example25()
{
```

```
    if(FrameListener)
    {
        delete FrameListener;
    }
}
```

7. 现在，在 ExampleApplication 类中创建一个名为 createFrameListener 的新函数。在这个函数中，创建一个 FrameListener 的实例并添加它到 mRoot:

```
void createFrameListener()
{
    FrameListener = new Example25FrameListener(_SinbadNode);
    mRoot->addFrameListener(FrameListener);
}
```

8. 编译运行程序。你应看到和之前相同的场景。但是这次，Sinbad 的实例向右边移动了并且你不能移动摄像机或用 Escape 键关闭程序。你可以使用控制台窗口的 X 按钮关闭窗口。如果你从控制台启动程序，你可以使用 CTRL+C 来结束程序。



刚刚发生了什么？

我们添加了一个可以移动我们场景结点的类到我们的代码中。

帧监听

我们在此遇到的一个新概念就是帧监听。顾名思义，帧监听是基于一个观察者的模式。我们可以添加一个类的实例。你可以使用 Ogre::Root 的 addFrameListener()方法，通过继承自 Ogre::FrameListener 的接口，添加一个对象到我们的 Ogre3D 根对象中。当对象被添加时，在确定的事件发生时我们的类就会得到通知。在这个例子中，我们重写了 frameStarted()方法。

在帧（帧，意思是指场景中的单个图像）被渲染之前，Ogre::Root 遍历所有被添加的 Framelistener 并调用其 frameStarted()方法。我们第四步中的函数定义中，我们沿 X 轴每帧 0.1 单位的速率变换结点。这个结点通过 Framelistener 的构造函数的传入。因此，场景每次被渲染的时候，结点就变换一点位置。结果，模型就移动了。正如我们在程序运行时看到的，我们不能移动摄像机或使用 Escape 键退出。这是因为这些是由 Framelistener 来操作的，而这个 FrameListener 是 ExampleApplication 框架自带的。ExampleApplication 是 SDK 自带的。现在我们代替其原有的 FrameListener，使用我们自己定义的实现过程。这样我们就不能使用自带的 Framelistener，但是这章我们将会重新实现他们中的大部分，所以不用担心。如果需要，我们可能仍然调用基类的成员函数来获得默认的行为。在第四步中，我们函数返回 true。如果返回 false，Ogre 3D 将会解释 false 为退出渲染循环的标志，而与此同时，程序就会结束。我们将会重新实现“按 Escape 键退出”的函数。

简单测试 —— FrameListener 的设计模式

1. Framelistener 概念的模式是基于?
   a. Decorator 【装饰者】
   b. Bridge 【桥】
   c. Observer 【观察者】

4.3 修改代码使其代替基于每帧的运动为基于时间的运动

根据您的电脑，场景中的模型可能移动的或快或慢或是刚刚好的速度。造成不同速度移动模型的原因是，在我们代码中，在每次渲染一个新帧之前，我们沿 Z 轴每帧移动模型 0.1 个单位。一个新的电脑可能以每秒 100 帧的速度渲染，那么模型将会每秒移动 10 个单位。当使用一个旧电脑的时候，可能就是 30 帧每秒，那么模型就只会移动 3 个单位。这相比新电脑只是不到三分之一。通常，我们想要程序在不同的平台和电脑性能下保持一致，这样他们就以相同的速度运行。这可以被 Ogre 简单的完成。

实践时刻 —— 添加基于时间的运动

我们将会使用之前的代码并指改变一行代码：

1. 改变我们变换结点的那行代码：

```
_node->translate(Ogre::Vector3(10，0，0) * evt.timeSinceLastFrame);
```

2. 编译运行程序。你可能会看到相同的场景，只有模型会以不同的速度移动。

刚刚发生了什么？

我们改变基于帧的运动模式为基于时间的模式。我们添加一个简单的乘法了实现基于时间的模式。正如前面所说，基于帧的运动有一些缺点。基于时间的运动更为出众，因为我们需要在所有的电脑上达到同样的运动效果而且在运动速度上有更高的可控性。在第一步，我们使用 Ogre 3D 的一个 *FrameEvent* 传递给调用的 *frameStarted()* 函数。这个 *FrameEvent* 类包含了在短时间内自从上一帧被渲染到现在的时间：

```
(Ogre::Vector3(10，0，0) * evt.timeSinceLastFrame);
```

这行代码使用这个公式来计算移动模型每帧移动的大小。我们使用一个三维向量乘以自上一帧到现在的秒数来计算。在当前的情况下，我们使用了向量(10，0，0)。这意味着我们想要模型每秒沿 X 轴移动 10 个单位。比如说，我们每秒渲染 10 帧;然后对于每一帧，*evt.timeSinceLastFrame* 将会是 0.1f。在每帧中，我们用向量(10，0，0)乘以 *evt.timeSinceLastFrame*，那么结果会是(1，0，0).这一结果将会应用于场景结点移动的每一帧。

简单测试 —— 基于时间和基于帧运动之间的不同

用你自己的话描述基于帧和基于时间运动的不同。

动手试试 —— 添加第二个模型

添加第二个模型到场景并使它沿场景中 Sinbad 模型的相反方向运动。

4.4 添加输入支持

我们现在已经有含移动物体的场景，但是我们想要程序可以像原来一样退出。因此，我们将会添加输入支持，并当 Escape 键按下的时候，我们可以退出程序。目前为止，我们只使用了 Ogre 3D；现在，我们将会使用 OIS (Object Oriented Input System[面向对象的输入系统])，OIS 是 Ogre 3D SDK 自带的，因为它被 ExampleFrameListener 所使用，但是在其他方面它是从 Ogre 3D 中完全独立出来的。

实践时刻 —— 添加输入支持

同样的，我们使用之前的代码并添加必要的代码以获得输入支持：

1. 我们需要添加一个新的参数到 Listener 的构造函数。

```
Example27FrameListener(Ogre::SceneNode* node，RenderWindow* win)
```

2. 当改变构造函数时，我们也需要改变实例化的地方：

```
Ogre::FrameListener* FrameListener = new Example27FrameListener(_
SinbadNode，mWindow);
```

3. 在这之后，我们需要添加代码到 listener 的构造函数。首先，我们需要两个辅助变量：

```
size_t windowHnd = 0;
std::stringstream windowHndStr;
```

4. 现在请求 Ogre 3D 获取渲染的窗口句柄：

```
win->getCustomAttribute("WINDOW"， &windowHnd);
```

5. 转换 handle 为一个 string：

```
windowHndStr << windowHnd;
```

6. 创建一个 OIS 参数表并添加窗口句柄：

（译者注：这章中参数表代表 ParamList，而 ParamList 是 OISPrereqs.h 中 std::multimap<std::string， std::string>的别名，而 multimap 请参考 STL 的容器。）

```
OIS::ParamList pl;
pl.insert(std::make_pair(std::string("WINDOW")， windowHndStr.str()));
```

7. 使用参数表创建输入系统：

```
_man = OIS::InputManager::createInputSystem( pl );
```

8. 然后创建一个 keyboard，这样我们就可以检查用户的输入了：

```
_key = static_cast<OIS::Keyboard*>(
                                    _man->createInputObject(OIS::OISKeyboard，  false ));
```

9. 我们创建的对象，必须在最后回收。添加一个析构函数到帧监听类中，这将会析构我们的 OIS 对象：

```
~Example27FrameListener()
{
   _man->destroyInputObject(_key);
   OIS::InputManager::destroyInputSystem(_man);
}
```

10. 在我们完成初始化之后，添加下面的代码到 *frameStarted()* 函数中的 node 变换之后，return 之前：

```
_key->capture();
if(_key->isKeyDown(OIS::KC_ESCAPE))
{
   return false;
}
```

11. 添加用户输入对象作为成员变量到帧监听类中：

```
OIS::InputManager* _man;
OIS::Keyboard* _key;
```

12. 编译运行程序。你现在应该可以看到程序可以通过按 Escape 键退出。

刚刚发生了什么？

我们创建了一个输入系统的实例并使用实例来获取用户的键盘输入。因为我们需要窗口的句柄来创建输入系统，我们改变帧监听类的构造函数，这样它就可以接受传递过来的一个渲染窗口的指针。这已经在第一步中完成了。我们然后从数值到字符串转换句柄并添加字符串到 OIS 的参数表。通过参数 m 表，我们可以创建我们输入系统的实例。

窗口句柄

一个窗口句柄仅仅是一个识别窗口的数值。这个数值是由操作系统创建并且每个窗口都有自己独一无二的句柄。输入系统需要这个句柄，因为没有它，输入系统就不能获取输入事件。Ogre 3D 为我们创建了一个窗口。所以为了获得窗口句柄，我们需要请求 Ogre 3D 执行下面一行代码：

```
win->getCustomAttribute("WINDOW",  &windowHnd);
```

渲染的窗口有多种属性，Ogre 3D 实现了一个通用 getter 函数。Ogre3D 也需要平台独立，因为每个平台都有自己的窗口句柄的变量类型，所以通用的函数是跨平台的唯一方法。这段代码中，WINDOW 是窗口句柄的标识符。我们需要传递传递一个指针来存贮句柄的值；在函数中这个指针将会被覆写。在我们接受句柄之后，我们使用 stringstream 来转换它为一个 string 类型，因为这是 OIS 所需要的。OIS 有同样的问题并使用同样的解决办法。在创建时，我们给 OIS 一个对组的参数表，这个对组是由一个标识字符串和一个字符串的值组成。

在第六步中，我们创建了这个参数表并添加字符串类型的句柄。在第七步使用这个参数表创建了输入系统。通过运用输入系统，我们可以在第八步中创建我们键盘接口。这个接口将会用于查询系统——用户按下的是哪个键？这一步将会在第九步中完成(译者:原作者应该指的是第十步)。每次在我们渲染帧之前，我们使用 capture() 函数来获取键盘的新状态。如果不调用这个函数，我们将不会得到键盘的新状态，并此因我们将永远得不到键盘事件了。在更新完状态之后，我们查询键盘是否现在按下 Escape 键。当按下的时候，我们了解到用户想要退出用户程序。这意味着我们必须返回 false 让 Ogre 3D 知道我们关闭程序。否则，如用户想要程序持续运行，我们可以返回 true 来使程序持续运行。

简单测试 —— 有关窗口的问题

什么是窗口句柄还有它是如何被程序和操作系统所使用的？

4.5 给模型添加动作

现在我们已经有能力获取用户的输入，现在让我们使用它来控制 Sinbad 的在平面上的运动吧。

实践时刻 —— 控制 Sinbad

我们使用以前的代码并添加代码到我们需要的地方，正如之前所做的一样。我们将会使用以下代码，使用 WASD 键来控制 Sinbad。

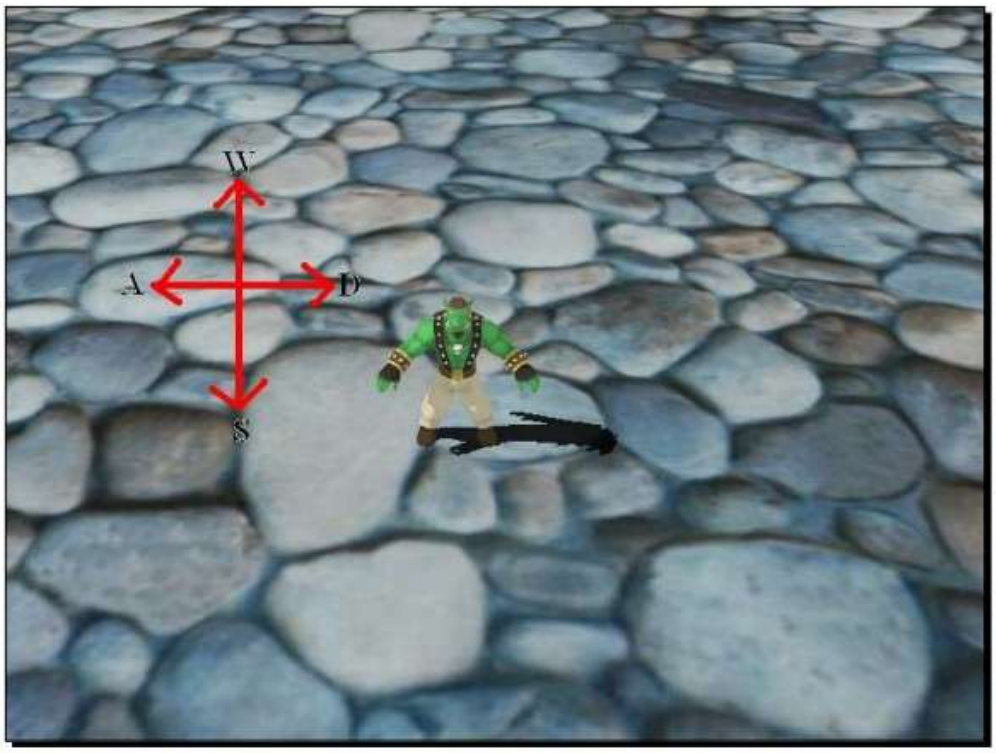
```
if(_key->isKeyDown(OIS::KC_W))
{
```

```
    translate += Ogre::Vector3(0，0，-10);
}
if(_key->isKeyDown(OIS::KC_S))
{
    translate += Ogre::Vector3(0，0，10);
}
if(_key->isKeyDown(OIS::KC_A))
{
    translate += Ogre::Vector3(-10，0，0);
}
if(_key->isKeyDown(OIS::KC_D))
{
    translate += Ogre::Vector3(10，0，0);
}
```

4. 现在使用向量来变换模型，并记住使用基于时间的运动而非基于帧的运动：

```
 _node->translate(translate*evt.timeSinceLastFrame);
```

5. 编译运行程序，然后你就可以用 WASD 键来控制 Sinbad 了。



刚刚发生了什么？

我们使用 WASD 键为用户添加基本的运动控制。我们查询所有的四个键并创建了存储运动的变量。我们使用基于时间的方式应用这个变量到模型。】

动手试试 —— 使用决定运动的速度因素

上面方法的缺点是我们想要改变模型的运动速度的时候，我们不得不改变四个向量。更好的方法是使用一个向量来表示运动方向，并使用一个 float 作为速度因子并使速度因子乘以变换向量。使用一个速度变量来改变代码。

4.6 添加一个摄像机

我们已经可以使用Escape并且我们可以移动Sinbad了。现在轮到我们的摄像机工作了。

实践时刻 —— 让我们的摄像机再次工作

我们已经创建了我们摄像机。现在我们将会让它和我们用户输入结合起来。

1. 扩展帧监听的构造函数，让它接受一个 camera 指针：

```
Example30FrameListener(Ogre::SceneNode* node，
                                        RenderWindow* win，Ogre::Camera* cam)
```

2. 同样添加一个成员变量来储存 camera 指针：】

```
Ogre::Camera* _Cam;
```

3. 然后添加参数到成员变量：

```
_Cam = cam;
```

4. 修改帧监听的初始化并添加 camera 指针：

```
Ogre::FrameListener* FrameListener = new Example30FrameListener(_
SinbadNode，mWindow，mCamera);
```

5. 我们需要获得鼠标的输入以移动摄像机。所以创建一个新的成员变量来存贮鼠标：

```
OIS::Mouse* _mouse;
```

6. 在构造函数中，在键盘操作后初始化鼠标：

```
_mouse = static_cast<OIS::Mouse*>(_man->createInputObject( OIS::OISMouse，  false ));
```

7. 现在我们已经有鼠标的状态了，我们也需要获取鼠标的状态。在获取键盘状态后面添加下面一行代码：

```
_mouse->capture();
```

8. 删去下面变换结点的代码：

```
_node->translate(translate*evt.timeSinceLastFrame * _movementspeed);
```

9. 在 frameStarted()函数中处理完键盘状态后，添加下面的代码来处理鼠标状态：

```
float rotX = _mouse->getMouseState().X.rel * evt.timeSinceLastFrame* -1;
float rotY = _mouse->getMouseState().Y.rel * evt.timeSinceLastFrame * -1;
```

10. 现在应用旋转和变换到摄像机：

```
_Cam->yaw(Ogre::Radian(rotX));
_Cam->pitch(Ogre::Radian(rotY));
_Cam->moveRelative(translate*evt.timeSinceLastFrame * _movementspeed);
```

11. 我们创建了一个鼠标对象，所以我们在帧监听的析构函数中销毁它。

```
_man->destroyInputObject(_mouse);
```

12 编译运行程序。你将会像之前一样操控场景了。】

刚刚发生了什么？

我们使用创建的摄像机同用户输入结合起来。为了控制摄像机，我们需要传递它到帧监听。我们在第一步和第二部中的构造函数中实现了这一点。我们也需要使用鼠标来控制我们的摄像机。所以第一我们不得不创建一个鼠标接口，这个已经在第六步中完成了，和我们创建键盘控制的方式一样。在第七步，我们调用了新鼠标接口的 capture() 函数来更新我们的鼠标状态。

鼠标状态

我们使用 isKeyDown() 函数来完成查询键盘状态。这次我们使用 getMouseState() 函数来完成查询鼠标状态的操作。这个函数返回一个鼠标状态作为 MouseState 类的实例，它包含了按钮按下与否和从上次获取鼠标状态开始鼠标是如何移动的按钮状态信息。

我们想要移动的信息是为计算摄像机需要旋转的大小。鼠标移动发生在了两个轴，即 X 轴和 Y 轴。两个轴的移动是由两个鼠标状态变量分开存贮的。我们然后就可以得到相对或绝对值。因为我们只对鼠标移动而非鼠标的位置的感兴趣，我们使用相对值。而绝对值包含包含鼠标在屏幕上的信息。当鼠标点进我们程序的确切区域的时候绝对值很是需要了。对于摄像机旋转，我们只需要鼠标移动信息，所以使用相对值。相对值只是描述出鼠标移动过的速度和方向是否改变，并不是描述所在位置的值。

这些值乘以自上一帧到现在的时间和-1。使用-1 是因为我们想让摄像机根据我们的需要进行旋转运动。所以我们需改变移动的方向。在计算完旋转的值，我们应用它们到 yaw() 和 pitch() 函数。最后一件事是应用我们从键盘输入创建的变换向量到摄像机。这个很有用，因为我们知道在局部空间中使用 (0，0，-1) 可以把摄像机向前移动。但当旋转应用于摄像机，这个就不一定正确了。请参考有关在三维空间中不同的空间的第二章寻找更为详尽的解释。

简单测试 —— 获取输入

为什么我们调用鼠标和键盘的 capture() 方法？

动手试试 —— 试玩一下例子

尝试从旋转计算中移除-1 并观察摄像机控制如何改变。

4.7 添加线框模式和点模式

在之前的第三章（摄像机，光源和阴影），我们使用 R 键来改变渲染模式为线框模式或点模式。现在我们添加这个特性到我们的帧监听。

实践时刻 —— 添加线框模和点渲染模式

我们使用刚创建的代码，并一如既往简单的添加的代码来实现我们想要的新特性：

1. 我们需要在帧监听中添加一个新的成员变量在保存现在的渲染模式：】

```
Ogre::PolygonMode _PolyMode;
```

2. 在构造函数中用 PM_SOLID 初始化创建的变量。

```
_PolyMode = Ogre::PolygonMode::PM_SOLID;
```

3. 然后我们在 *frameStarted()*中添加一个新的函数，来测试 R 键是否按下。如果情况是这样，我们可以改变渲染模式。如果目前的模式为实体模式，我们想要它成为线框渲染模式。

```
if(_key->isKeyDown(OIS::KC_R))
{
   if(_PolyMode == PM_SOLID)
   {
     _PolyMode = Ogre::PolygonMode::PM_WIREFRAME;
   }
```

如果它是线框型的，我们想要改变为点模式：

```
   else if(_PolyMode == PM_WIREFRAME)
   {
     _PolyMode = Ogre::PolygonMode::PM_POINTS;
   }
```

4. 如果它是点模式，最后让它变回实体模式：

```
   else if(_PolyMode == PM_POINTS)
   {
     _PolyMode = Ogre::PolygonMode::PM_SOLID;
   }
```

6. 现在我们已经设置了新的渲染模式，我们可以应用它并关闭 if 语句了：

```
_Cam->setPolygonMode(_PolyMode);
}
```

7. 编译运行程序；你应会看到按过 R 键后，渲染模式改变：

刚刚发生了什么？

我们使用了函数 *setPolygonMode()* 来改变实体模式为线框模式，然后是点模式。我们

总是保存最后一个值。这样当改变模式时，我们知道当下的模式时什么，我们将会改变为什么模式。我们从实体模式改变为线框模式到点模式，然后又变为实体模式。我们注意到，当按下 R 键时，渲染模式改变的相当迅速。这个因为我们每一帧都检查 R 键是否犯下并且相对于计算机的每帧的速度，人类按下 R 键的速度是很慢的。这导致，我们的程序会认为我们在短时间内按下多次 R 键，这就意味着不定的哪帧将会切换到线框模式。这不是最理想的情况，但存在一种办法让我们做的的更好，我们下面将会看到。

4.8 添加一个计时器

解决渲染模式改变过快的一种解决办法就是使用计时器。每一次我们按下 R 键，一个计时器就启动了，并且只有当足够的时间过去，我们才能处理另一个 R 键的按下。

实践时刻 —— 添加一个计时器

1. 添加一个计时器作为帧监听的成员变量：

```
Ogre::Timer _timer;
```

2. 在构造函数中复位计时器：

```
_timer.reset();
```

3. 现在添加一个来检查自从上次按下 R 键是否有 0.25 秒过去：

```
if(_key->isKeyDown(OIS::KC_R) && _timer.getMilliseconds() > 250)
{
```

4. 如果过去足够的时间，我们需要复位。否则，R 键只能按下一次：

```
_timer.reset();
```

5. 编译运行程序；当现在按下 R 键，它将改变渲染模式为下一个。

刚刚发生了什么？

我们使用了 Ogre 3D 的另一个新类，即是 Ogre::Timer。顾名思义，这个类提供了计时器的功能.我们在帧监听的构造函数中复位了计时器，并在每次用户 R 键，我们会检查自动上次调用 reset()函数，0.25 秒是否过去。如果是这种情况，我们输入 if 的代码段，复位计时器而且像之前一样的方式改变需渲染模式。这必须保证需渲染模式时在 0.25 秒之后改变

的。当我们一直按着 R 键，我们看到程序在每次 0.25 秒的等待之后，将会改变渲染模式。】

动手试试 —— 改变输入模式

通过改变代码使渲染模式不是在一个确定的时间过去才可改变，只有当释放 R 键并再次按下的时候才改变。

4.9 小结

在这一章，我们学习了帧监听的借口和如何使用它们。我们也学习了如何启动 OIS，在这之后，我们学习了如何查询键盘和鼠标的状态：

具体的，我们学习了：

* 当新一帧被渲染的时候，如何得到通知。
* 基于帧和基于时间运动的重要不同。
* 如何使用用户输入实现我们的摄像机移动。
* 如何改变摄像机的渲染模式。

现在我们已经实现了我们帧监听的基础函数，我们将在下一章学习如何运动模型。

第五章 使用 Ogre 3D 动画模型

这章的重点会在模型动作和它们通常是怎样工作上，特别是在 Ogre 3D 方面。没有动画，那么一个 3D 场景是没有生机的。动画是使场景现实和有趣的最重要的因素之一。

在这一章，我们将会：

* 播放一个动画。
* 把两个动画结合起来。
* 在动画上关联实体。

5.1 添加动画

在之前的章节中，我们使用用户输入添加了程序的交互性。现在我们准备添加动画来增加另一种形式的交互性。动画对于每个场景是非常重要的因素。没有它们，每个场景看起来将会是静止的而且死气沉沉的，但是加入动画后，场景将会生动起来。所以让我们添加它们吧。

实践时刻 —— 添加动画

一如往常，我们将会使用之前章节的代码，但是这次我们什么也不删除。

1. 对于我们的动画，我们需要在帧监听中添加新的成员变量。添加一个指针来储存我们想要动画的实体，另一个指针保存动画状态：

```
Ogre::Entity* _ent;
Ogre::AnimationState* _aniState;
```

2. 然后改变帧监听的构造函数把实体的指针作为一个新的参数：

```
Example34FrameListener(Ogre::SceneNode* node，Ogre::Entity*
ent，RenderWindow* win，Ogre::Camera* cam)
```

在构造函数的函数体中，把参数的实体指针赋值给新的成员变量：

```
_ent = ent;
```

4. 在这之后，从实体中调用 Dance 动作并存储到专门为它而创建的成员变量中。最后，设置动画为 enabled 并循环动画。

```
_aniState = _ent->getAnimationState("Dance");
_aniState->setEnabled(true);
_aniState->setLoop(true);
```

5. 然后，我们需要告诉动画自从上一次更新到现在已经过去多长时间。我们将会在 frameStarted() 函数中添加这件事；通过这个我们就可以。

```
_aniState->addTime(evt.timeSinceLastFrame);
```

6. 我们最后需要做的就是调整 ExampleApplication 类让它可以与新的帧监听一起作用。添加一个新的成员变量到程序来保存一个实体指针。

```
Ogre::Entity* _SinbadEnt;
```

7. 把原来创建的局部指针分配给新创建的实体来储存。把

```
Ogre::Entity* Sinbad = mSceneMgr->createEntity("Sinbad"， "Sinbad.mesh");
```

替换为

```
_SinbadEnt = mSceneMgr->createEntity("Sinbad"， "Sinbad.mesh");
```

8. 当关联实体到结点也需要做同样的改变。

```
_SinbadNode->attachObject(_SinbadEnt);
```

9. 当然，当创建新的帧监听时，添加新的参数到调用的构造函数。

```
Ogre::FrameListener* FrameListener = new Example34FrameListener(_
SinbadNode，_SinbadEnt，mWindow，mCamera);
```

10. 编译运行程序。你将会看到 Sinbad 在跳舞。】

刚刚发生了什么 ？

用短短几行代码，我们实现了 Sinbad 的跳动。在第一步中，我们添加两个成员变量，用于动画模型。第一个指针变量用户存储模型。第二个 Ogre::AnimationState 的指针变量，被 Ogre 3D 用于描述单一动画和其相关联的信息。第二和第三步就比较容易理解了，第二步我们改变了构造函数来适应新指针，第三步保存了第一步中创建的成员变量。第四步就比较有趣了，我们请求实体返回给我们名为”Dance”的动画。每个实体变量保存了实体本身所有的动画，并且我们可以使用字符串变量和 getAnimationState() 函数查询动画。这个函数返回动画的 AnimationState 指针，如果动画为空，它将会返回一个空指针。在获取完动画的状态，我们激活它

这告诉 Ogre 3D 播放这个动作。同样，我们设置循环属性为真，这样这个动作就可以循环播放，直到我们停止它为止。第五步是重要的一步，使用这行代码，使实体栩栩如生起来。每次场景被渲染时，然后添加动画时间，这样 Ogre 3D 就可以播放了。确切点说，它相应自上一帧到现在的过去时间。这可能要由 Ogre 3D 自己完成，但是这种方式更灵活。比如，我们可以添加第二个模型，但是我们想要它做个慢动作。如果 Ogre 3D 自己更新动作，那么不论是以普通速度还是以慢速动画，模型的动画对我们来说将是很困难的。但是我们自己完成，我们可以把 evt.timeSinceLastFrame *0.25 便可以实现慢动作。在这步之后，便可以对构造函数进行小的修改，使之与帧监听的构造函数兼容。这样，我们需要保存想要动画的实体指针。

简单测试 —— 时间的重要性

为什么我们需要告诉 Ogre 3D 自从上次更新到现在时间过去了多少？这种方式的优点是什么？

动手试试 —— 添加第二个模型

添加第二个模型，这个模型站在第一个模型旁边，但是以一个慢动作在动画。你应看到两个模型在显示动画的不同阶段，就好像下面图片显示的那样：

5.2 同时播放两个动画

在添加我们第一个动作之后，我们将会研究为什么和怎样同时播放动作。

实践时刻 —— 添加第二个动作

这里，我们将会使用与创建第一个例子同样的办法添加第二个动作：

1. 从 Dance，改变动画到 RunBase。

```
_aniState = _ent->getAnimationState("RunBase");
```

2. 编译运行程序。你可以看到 Sinbad 在跑动，但只是上半身的不动的原地跑动。

3. 对于我们第二个动作，我们需要一个保存动画状态的新指针：

```
Ogre::AnimationState* _aniStateTop;
```

4. 然后，我们需要获取动画的状态，激活并循环动画。我们需要调用的动画名为 Runtop。

```
_aniStateTop = _ent->getAnimationState("RunTop");
_aniStateTop->setEnabled(true);
_aniStateTop->setLoop(true);
```

5. 最后要做的一件事就是添加自上帧过去的时间，就好像我们第一次设置的那样：

```
_aniStateTop->addTime(evt.timeSinceLastFrame);
```

6. 然后编译运行程序。现在你可以看到 Sinbad 整个身子做跑的动作。

刚刚发生了什么 ？

我们在同时使用了两个动画。在之前，如果你问自己为什么不调用像playAnimation(AnimationName)这样的函数，反而需要获取 AnimationState 来播放动画。现在你有答案了。Ogre 3D 支持在同一时间使用多种动画，但使用playAnimation(AnimationName)却实现不了。我们甚至可通过使用一个修改的变量和addTime()函数，以不同速度来控制模型。

动手试试 —— 添加控制动画速度的因素

添加控制 Sinbad 上半身动画因素，尝试赋以 0.5 或 0.4，并看下这个动画的效果。

5.3 让我们的模型走两步

我们已经有个行走的动画了，但是模型并没有改变它的位置。我们将会添加基础的模型动画控制并使用我们所学来混合动画。】

实践时刻 —— 用户控制和动画相结合

然后，一如既往，我们将会使用之前的代码来作为起点：

1. 首先，我们需要帧监听中两个变量来控制移动和保存旋转。

```
float _WalkingSpeed;
float _rotation;
```

2. 在构造函数中，我们初始化新的变量；我们每秒移动 50 个单位并以无旋转作为开始：

```
_WalkingSpeed = 50.0f;
_rotation = 0.0f;
```

3. 然后我们需要改变动画状态并阻止它循环。这次，当新的动画开始时，我们需要控制模型而非交给 Ogre 3D 控制。

```
_aniState = _ent->getAnimationState("RunBase");
_aniState->setLoop(false);
_aniStateTop = _ent->getAnimationState("RunTop");
_aniStateTop->setLoop(false);
```

4. 在 *frameStarted()* 方法中，我们需要两个局部变量，一个是为了显示这帧我们是否移动了模型，第二个变量是用于保存模型移动的方向。

```
bool walked = false;
Ogre::Vector3 SinbadTranslate(0，0，0);
```

5. 同样在方法中，我们添加新的代码来控制模型的移动。我们将会使用方向键来移动。当一个键按下的时候，我们需要保存变换变量来存储模型移动的方向，并且需要以它移动的方向来设置旋转变量以旋转模型

```
if(_key->isKeyDown(OIS::KC_UP))
{
    SinbadTranslate += Ogre::Vector3(0，0，-1);
    _rotation = 3.14f;
    walked = true;
}
```

6. 我们需要以同样的方式添加另外三个方向键：

```
if(_key->isKeyDown(OIS::KC_DOWN))
```

```
{
    SinbadTranslate += Ogre::Vector3(0，0，1);
    _rotation = 0.0f;
    walked = true;
}
if(_key->isKeyDown(OIS::KC_LEFT))
{
    SinbadTranslate += Ogre::Vector3(-1，0，0);
    _rotation = -1.57f;
    walked = true;
}
if(_key->isKeyDown(OIS::KC_RIGHT))
{
    SinbadTranslate += Ogre::Vector3(1，0，0);
    _rotation = 1.57f;
    walked = true;
}
```

7. 然后，在按键处理之后，我们需要检查一下这帧中 walked 是否为 true。如果是这种情况，我们需要检查动画是否停止。当为 true 时，我们重新开始动画：

```
if(walked)
{
    _aniState->setEnabled(true);
    _aniStateTop->setEnabled(true);
    if(_aniState->hasEnded())
    {
        _aniState->setTimePosition(0.0f);
    }
    if(_aniStateTop->hasEnded())
    {
        _aniStateTop->setTimePosition(0.0f);
    }
}
```

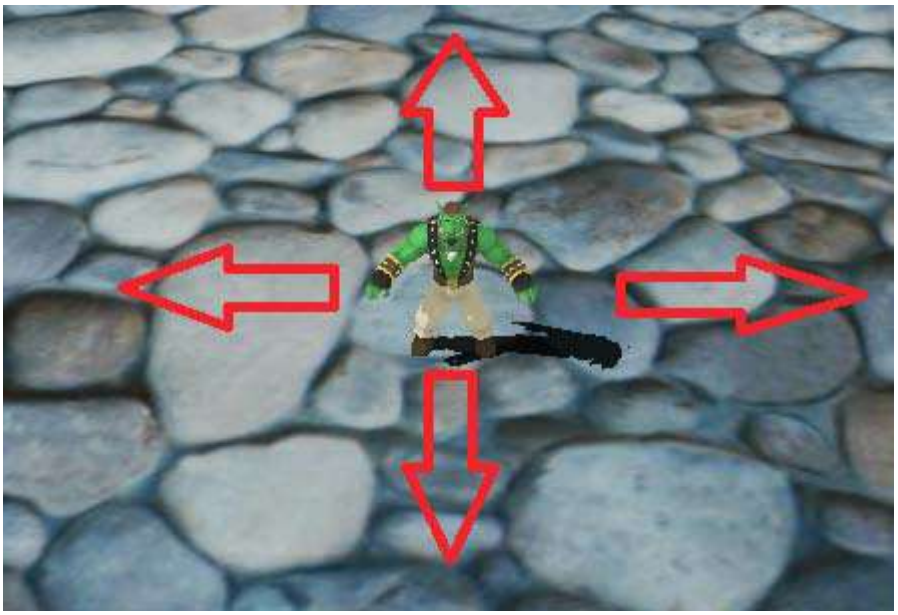
8. 如果我们这帧不动的话，需要设置两个动画的状态到 0。否则，我们的模型将会看起来像在进行动作的一半时被冻住一般，而这看起来的效果不好。所以如果我们这帧不需要行走，我们需设置两个动画回到其起点位置。同样，当这帧不移动模型的时候，我们将会冻结两个动画，因为此时我们并不需要动画。

```
else
{
   _aniState->setTimePosition(0.0f);
   _aniState->setEnabled(false);
   _aniStateTop->setTimePosition(0.0f);
   _aniStateTop->setEnabled(false);
}
```

9. 最后我们需要做的就是应用变换和旋转到模型的场景结点：

```
_node->translate(SinbadTranslate * evt.timeSinceLastFrame * _ WalkingSpeed);
_node->resetOrientation();
_node->yaw(Ogre::Radian(_rotation));
```

10 现在我们编译运行程序。在鼠标和 WASD 键的作用下，我们可以移动摄像机。用方向键，我们就可以移动 Sinbad。每次移动他的时候，他便会做出正确的动画。



刚刚发生了什么 ？

我们综合了用户输入和动画创建了第一个程序。这可以称得上我们目前为止第一个真正的交互程序。在第一步和第二步，我们创建和初始化了我们需要的变量。在第三步，我们改变了过去动画的方式；准确的说，我们之前总是直接激活动画并循环它。现在我们不想要直接激活它，因为我们只需要在移动的时候使用动画，除此之外的情况，看起来都很stupid。这也是我们冻结动画的循环的原因。我们只想要反应用户的输入，这样就没必要一直循环了。如果需要，我们将会自己启动动画。

我们基本上是在 frameStarted() 方法中作出的改变。在第四步中，我们创建了一对稍后使用的局部变量。一个是来表示这帧模型是否移动 bool 类型的开关，另一个是表示移动方向的向量。第五步和第六步中查询方向键的状态。当一个键按下，我们改变方向向量，做相应的旋转并设置移动的开关为 true。在第七步中我们使用这个开关，如果 flag 为 true，意味着这帧模型会移动，我们激活动画并检查是否有动画达到它们的结尾。如果动画到达结尾，我们重置它们到起点这样，它们就可以再次继续播放了。因为当模型不移动的时候，我们不想要播放动画，所以在第八步我们设置它们动画为起点并冻结它们。在第九步中，我们应用了变换。然后重置旋转，在这之后，应用新的旋转。这一步很是必要，因为 yaw 函数添加旋转到现有的旋转，我们需要绝对的旋转而不是相对的旋转。因此，我们先重置旋转然后应用旋转到现在的归零旋转上。

添加双刀

我们现在有一个可以通过用户输入控制行走和动画的模型。现在我们准备研究我们如何添加一个对象到动画模型上。

1. 在createScene函数之后，创建两把刀模型的实例并命名为 Sword1 和 Sword2：

```
Ogre::Entity* sword1 = mSceneMgr->createEntity("Sword1"， "Sword.mesh");
Ogre::Entity* sword2 = mSceneMgr->createEntity("Sword2"， "Sword.mesh");
```

2. 现在使用一个名称来关联刀到模型：

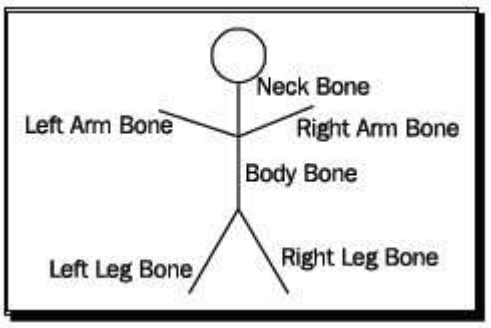
```
_SinbadEnt->attachObjectToBone("Handle.L"， sword1);
_SinbadEnt->attachObjectToBone("Handle.R"， sword2);
```

3. 编译运行程序。你将会看到 Sinbad 手里有两把刀。

刚刚发生了什么？

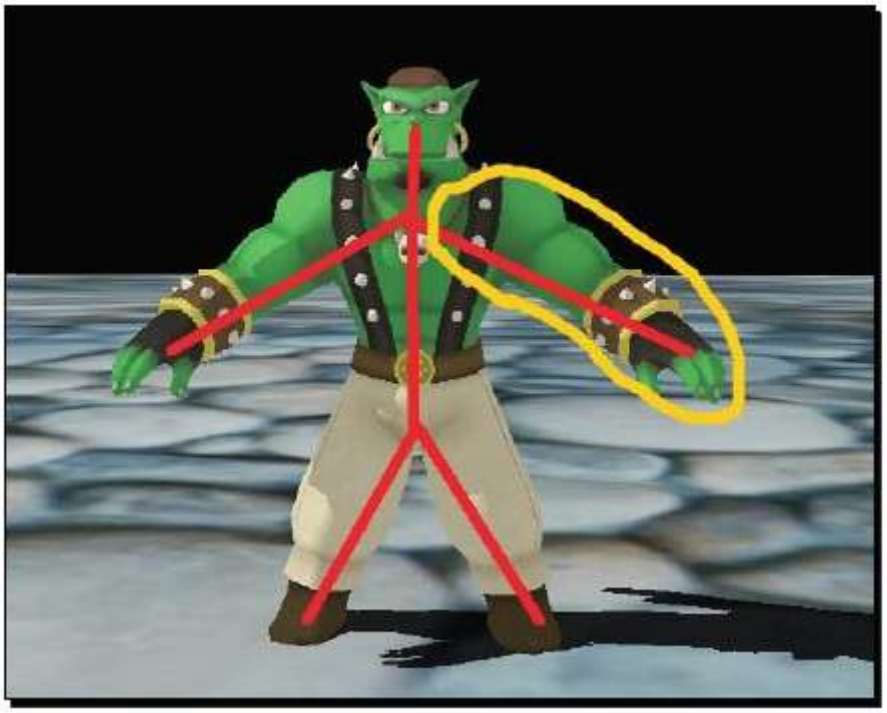
我们创建了两把刀的实例并关联它们到骨骼上。创建的实例不能难以理解。刚才的代码中比较难但比较有趣的部分就是调用了函数 attachObjectToBone()。为理解这个函数的工作原理，我们需要讨论一下动画是如何保存和播放的。

动画



对于动画来说，我们使用一种称为 skeletons(骨架) 和 bones(骨骼) 的东西。这个系统的灵感是来自于自然；在自然界，基本上所有的生物都有一个骨架来支撑它。在骨架和一些肌肉的帮助之下，动物和人类可以朝一个确定的方向移动他们身体的一部分。比如，我们可以使用手指的关节来把手指握成一个拳头。在计算机图形学中动画也是以同样的方式来运

行。美工定义 3D 模型而且为其创建骨架，这样模型就可以动画了。骨架是由骨骼和关节构成。关节是连接两块骨骼并且定义骨骼的动画方向。在这里是简化的骨架图；而通常，要比这里的多很多。



用关节，骨骼和骨骼的作用半径，一个美工就可以创建复杂的动画，比如我们正在使用的 Sinbad 动画。就好像动画，骨骼也是有名字的。Ogre 3D 让我们使用这些骨骼作为关联别的实体的点。这有着巨大的优势，当关联实体后，实体也会像骨骼一样得到旋转，这意味着如果我们在 Sinbad 的手中有一个关联点并关联刀之后，双刀就会一直在手中。因为当双手做动作时，刀也会得到同样的动画。如果这个函数不存在，那几乎不可能添加模型到手上或者关联东西到他们的后面，就如我们对刀做的操作一样。】

5.4 打印输出模型的所有动画

我们已经知道美工定义动画的名字，但很多时候直接从 Ogre 3D 来获取动画的名字很重要。当我们没有制作这个模型的美工来询问里面有什么动画或当你想要检查一下导出的处理是否成功，获取名字这件事就显得十分重要了。现在我们将会研究如何在控制台中打印模型的所有动画。

实践时刻 —— 打印所有的动画

我们将会使用之前的代码作为起点来打印我们实体所有的动画：

1. 在 *createScene()* 函数最后，用set获取模型所有的动画：

```
Ogre::AnimationStateSet* set = _SinbadEnt- >getAllAnimationStates();
```

2. 然后定义一个迭代器并用一个set迭代器来初始化：

```
Ogre::AnimationStateIterator iter = set- >getAnimationStateIterator();
```

3. 并且，最后，遍历所有的动作并打印它们的名字：

```
while(iter.hasMoreElements())
{
    std::cout << iter.getNext()->getAnimationName() << std::endl;
}
```

4. 编译运行程序。在开始程序并加载场景之后，你将会在控制台程序中看到下面的文字。

Dance
DrawSwords
HandsClosed
HandsRelaxed
IdleBase
IdleTop
JumpEnd
JumpLoop
JumpStart
RunBase
RunTop
SliceHorizontal
SliceVertical

刚刚发生了什么？

我们请求实体返回给我们包含动画信息的数据集。我们然后遍历这个数据集并打印出动画的名称。我们看到有好多没有用过的和已经用过的动画。

5.5 小结

在这章我们学习了很多有关动画和如何使用它来使我们的3D更有趣的方法。

具体的说，我们所学涵盖了以下内容：

* 如何获取实体的动画并使用它。
* 如何激活，冻结和循环程序和为什么需要告诉动画自上帧过去有多长时间。
* 如何在同一时间使用两个动画。
* 动画是如何使用骨架的？如何关联一个实体到一骨骼上？
* 如何查询实体包含的所有动画。

在下一章，我们将会研究 Ogre 3D 的另一个新方面，主要是关于使用不同的场景管理器和为什么要这样做。

第六章 场景管理器

Ogre 提供了很多很多的功能。在这章，我们将会接触一些我们之前没有用过的技术，但是有一些在创建复杂的3D 场景中很有帮助，比如SceneManagers，创建我们自己的模型，加速我们的程序和有效率的处理大量的3D 数据

在这章，我们将会：

* 学习如何改变现有的场景管理器。
* 学习什么是 Octree。
* 学习用代码创建自己的实体。
* 学习使用静态集合加速我们的程序。

6.1 以空的程序作为起点

这次我们将会使用几乎空白的程序并以这个程序作为起点

实践时刻 —— 创建新的程序

1. 首先我们需要引入 ExampleApplication 头文件：

```
#include "Ogre\ExampleApplication.h"
```

2. 创建一个从ExampleApplication继承而来的新类并创建一个空的 *createScene()* 函数：

```
class Example41 : public ExampleApplication
{
public:
    void createScene()
    {
    }
};
```

3. 最后，我们需要一个main函数来创建程序的实例并运行它。

```
int main (void)
{
   Example41 app;
   app.go();
   return 0;
}
```

4. 以同样的头文件和之前使用库目录来编译项目。你将会得到一个可以按 escape 建关闭的黑色窗口。

刚刚发生了什么？

我们创建了一个从 ExampleApplication 继承的类。它有一个空的 createScene 函数在基类中它是一个纯虚函数，而且如果我们不重写它，我们不能建立类的实例。我们程序的最有趣的部分将会从现在开始。

6.2 获取场景管理器的类型

下面，我们将添加一些代码来打印我们正在使用的场景管理器类型。

实践时刻 —— 打印输出场景管理器的类型

我们使用之前的代码。一如既往的，也需要添加后面的代码:】

1. 使用 *createScene()* 函数打印出场景管理器的名字：

```
std::cout << mSceneMgr->getTypeName() << "::" << mSceneMgr->getName() << std::endl;
```

2. 编译运行程序。当程序开始在控制台运行时，你可以看到下面一行：

OctreeSceneManager::ExampleSMInstance

刚刚发生了什么？

我们添加了一行可以打印出场景管理器名字和类型的代码。在当前情况下，场景管理器的名字是 ExampleSMInstance，这清楚的表示了在示例程序的场景中使用的管理器的名称。SM 代表 Scene Manager（场景管理器）。最有趣的部分是类型，在当前情况下是八叉树场景管理器(OctreeSceneManager).在我们详细讲解什么是 OctreeSceneManager 之

前，让我们讨论一下在 Ogre 3D 中 场景管理器一般做什么？

场景管理器是用来干什么的？

场景管理器做着很多的事情，当去查看 Ogre 的文档时，我们就会清楚的发现这一点。管理器中有大量关于创建，销毁，获取，设置和赋值的函数。我们已经使用了一些函数，比如 createEntity()，createLight() 和 getRootSceneNode()函数。场景管理器的重要任务之一就是实现对象的管理。这些对象可能是场景结点，实体，光源或者其他一些 Ogre 3D 的对象。场景管理器就扮演一个制造和摧毁对象的工厂。Ogre 3D 工作的原则是——谁创建，谁摧毁。每次我们想要删除一个实体或场景结点的时候，我们必须使用场景管理器。否则 Ogre 3D 可能会尝试稍后自动释放内存，这可能会导致程序的崩溃。

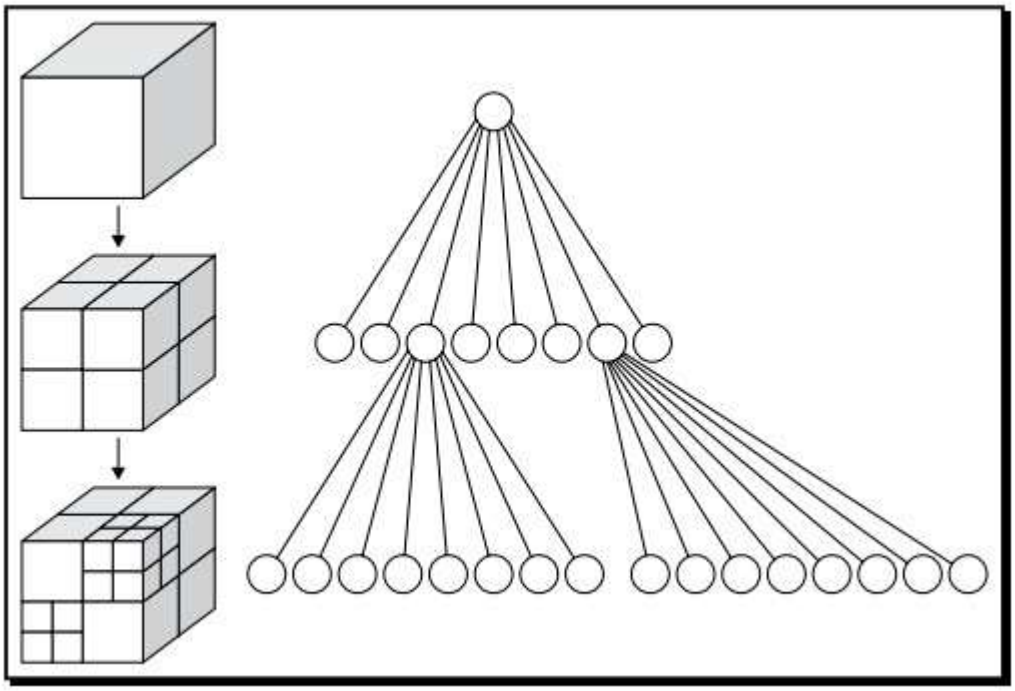
场景管理器除了管理对象，也如它名字所暗示的那样管理场景。这可能包含优化场景和计算每个对象是在场景中渲染的位置。它会实现高效的裁剪运算。每次我们移动一个场景结点，它会标志为已移动。当场景被渲染的时候，移动结点和其子结点的位置将会被计算出来。至于其它，我们使用上一帧的位置。这就节省了大量的计算时间而且这是场景管理器的重要任务之一。

为方便拣选的目地，我们需要一个快速的方法来丢弃摄像机渲染过程中不可见的部分。这意味着，我们需要一个简便的办法来遍历场景和测试结点的可见性。有不同的算法来实现这个目地，Ogre3D 带有不同的场景管理器，并且用多种算法来实现多种场景类型。

我们一直使用的是 OctreeSceneManager。这个场景管理器名字是由来是使用一个八叉树来存储场景。那么什么又是 Octree 呢？

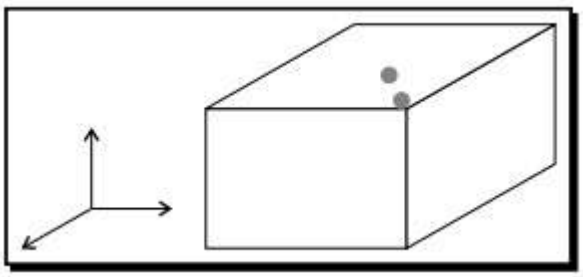
八叉树

顾名思义，八叉树是一种树形结构。想每个树形结构一样，它有一个根而且每个子结点都有一个父节点。和普通树形结构不同是的每个结点最多有八个子结点，故名 Octree（八叉树）。下面的图示就展示出了一个 Octree（八叉树）。

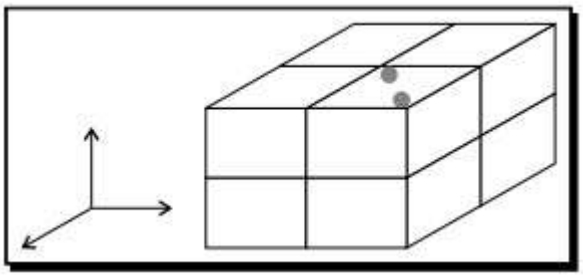


资料来源：http://commons.wikimedia.org/wiki/File:Octree2.svg

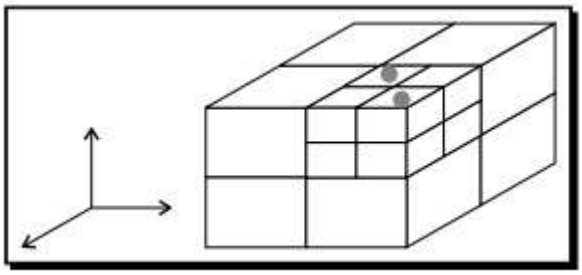
但为什么 Ogre 3D 使用一个八叉树来存储 3D 场景？一个八叉树有一些极其有用的属性来存储 3D 场景。其中之一就是它有多达 8 个的子结点。比如，如果我们拥有一个有两个对象的 3D 场景，我们可以用一立方体来封闭场景。



如果我们把这个立方体按长，宽，高的一半来分来，我们可以得到 8 个新的立方体，每个封闭了 1/8 的场景。这八个立方体可以别认为是原立方体的八个子结点。



现在两个场景的对象在立方里的右上前端了。其他七个立方体都是空的，因此现在省略他们。我们将会把含有两个对象的立方体再次分割。



现在每个立方体封闭这一个或者零个对象，他们都是叶子结点。八叉树的这个树形使得拣选变得简单而快速。当渲染一个场景的时候，我们在场景中有一个摄像机，视野为一视椎的面积。我们以八叉树的根结点作为开始来决定渲染哪个对象。如果视椎与立方体相交叉，我们继续遍历子结点。以上情况总是发生，因为开始时摄像机拍摄整个封闭场景的时候，八叉树从 0 开始索引。然后接着索引八叉树到下一层，也就是根结点的子结点。我们每次对子结点进行视椎体拣选测试，如果它们相交，我们继续索引立方体的子立方体。同样，如果立方里完全在视椎体内，我们不需要进行深度的遍历，因为我们知道立方里内的所有子立方体也都在视椎体内。一直这样做直到遍历到叶子为止，然后继续遍历另一个子结点，也直到叶子为止。通过这种算法，我们每步都可以去掉大部分的渲染场景，并且在几步之后，我们或者空的叶子树或只有一个叶子的对象。然后就可以知道那些对象是可视的渲染对象。这种算法的妙处就是可以在开始几步就可以去掉大部分的场景。假如在我们的例子中，我们整个视椎体看到两个对象在深度为 1 的立方体中，我们就可以在第一步去掉 8 分之 7 的场景。

这种处理方式类似于二叉树。它们之间的不同就是前者可有八个子结点，而后者只可有两个子结点。

6.3 另一种场景管理器类型

我看已简单研究第一种场景管理器了。现在让我们来看下另一种。

实践时刻 —— 使用另一种场景管理器

我们再次使用之前 example 中的代码：

1. 删去 *createScene()* 函数中所有的代码。

2. 添加一个名为 *chooseSceneManager()* 的新函数到程序的类中：

```
virtual void chooseSceneManager(void)
{
}
```

3. 现在添加代码到新的函数中以加载一个包含我们想要的地图的文件。

```
ResourceGroupManager::getSingleton().addResourceLocation(
                        "../../media/packs/chiropteraDM.pk3"，  "Zip"，
                        ResourceGroupManager::getSingleton().
                        getWorldResourceGroupName()，  true );
```

4. 在添加完地图之后，我们需要把完整地图的信息它加载进去：

```
ResourceGroupManager::getSingleton().initialiseResourceGroup(
                    ResourceGroupManager::getSingleton().getWorldResourceGroupName());
```

5. 然后我们需要使用 createSceneManager() 函数：

```
mSceneMgr = mRoot->createSceneManager("BspSceneManager");
```

6. 现在告诉场景管理器我们想要显示之前加载的地图：

```
mSceneMgr->setWorldGeometry("maps/chiropteradm.bsp");
```

7. 编译运行程序。你会看到一个取自一个著名游戏的地图并且你可以操控方向穿越地图。但是因为这个游戏使用了不同 up 向量，所以旋转有些不同并且浏览起来也感觉有点怪异。

刚刚发生了什么？

我们使用了 chooseSceneManager() 函数来创建一个不同于默认的场景管理器。在这种情况下，我们创建了 BspSceneManager（BSP 场景管理器）。BSP 代表二叉空间分割，它是一种被多年前的老式射击游戏用来存储信息的技术。BSP 分割层为多个凸部分并把它们按树形结构储存起来。在老式的显卡上，这使渲染和别的图形任务的执行变得更快。而今，BSP 场景已经没有多年前使用的频繁了。

资源管理器

代码中的第一行使用我们从未使用的称为 ResourceGroupManager(资源管理器)的管理器。这种管理器在我们程序的生命周期中一直加载着各种资源。在启动期间，资源管理器获取到资源的目录列表和我们想要加载的 zip 压缩包。这个目录可以从文件中读取，比如 resources.cfg，或者也可以把文件写进程序代码。在这之后，我们就可以仅仅使用文件名来创建实体，而不需要文件的整个路径，因为管理器已经索引过了。只有当我们创建索引文件的实例的时候它才会真正的被加载进来。索引帮助我们避免了加载同样模型的两次的检查。但我们使用模型两次，管理器仅加载模型一次，并且当需要两个同样模型的实例的时候，管理器使用已经加载过的模型并且不会再次加载。

addResourceLocation() 函数获取一个文件夹或 zip 压缩包的路径，第二个参数定义它的类型，通常它可以是 zip 压缩包或者一个文件夹。如果需要，我们可以添加自己的资源的类型。当我们想要加载自己定义的数据包类型时，这会变得非常有用。

第三个参数是我们想要加载文件到所在资源组的名称。资源组就好像 C++的命令空间一样；因为我们加载的是一张游戏地图的一部分，是加载预先定义好的被 WorldResourceGroup 返回的资源组名称。最后一个函数告诉 Ogre 3D ，我们加载的路径是否相被递归调用。如果设置为 false，只有在目录中的文件被加载，在子文件夹的文件不被加载。如果设置为 true，Ogre 3D 也会加载子文件的文件。默认的设置为 false。

通过调用 initialiseResourceGroup()函数，我们告诉 Ogre 3D 去索引在 ResourceGroup 没有索引到的文件。当然，我们必须添加想要索引的资源组。在这个调用完之后，我们可以使用关联到此资源组的所有文件。

setWorldGeometry

setWorldGeometry() 是个特殊的函数告诉 BspSceneManager 去加载保存在 bsp 文件类型中的地图。对于一个地图，我们使用保存在 .pk3 为后缀的 BSP 文件——这就是为什么我们需要在第一步中加载压缩包的原因。

6.4 创建我们自己的模型

我们已经看到如何使用不同的场景管理器和如何使用一个场景管理器加载地图。现在我们将会研究如何只在 plane 类的帮助下用代码创建一个 mesh。这次，我们将会自己做所有的

事情。我们将会创建一个在地面上有草地的模型。

实践时刻 —— 创建一个显示一片草的模型

这次，我们需要使用 OctreeSceneManager，所以我们不需要使用 chooseSceneManager() 函数了：

1. 我们需要一个空的程序：

```
class Example43 : public ExampleApplication
{
private:

public:
   void createScene()
   {

   }
};
```

2. 我们需要在 *createScene()* 函数中首先使用的是一个平面的定义。我们将会使用这个平面作为我们放置草的地面：

```
Ogre::Plane plane(Vector3::UNIT_Y，  -10);
Ogre::MeshManager::getSingleton().createPlane("plane"，
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME，
                    plane，1500，1500，200，200，true，1，5，5，Vector3::UNIT_Z);
```

3. 然后实例化我们刚刚创建的平面并给它设置一种材质。我们将会使用从 expamles 中加载的 GrassFloor 材质：

```
Ogre::Entity* ent = mSceneMgr->createEntity("GrassPlane"，  "plane");
mSceneMgr->getRootSceneNode()->createChildSceneNode()->attachObject(ent);
ent->setMaterialName("Examples/GrassFloor");
```

4. 然后添加方向光到场景。否则，场景过暗就什么都看不到了：

```
Ogre::Light* light = mSceneMgr->createLight("Light1");
light->setType(Ogre::Light::LT_DIRECTIONAL);
light->setDirection(Ogre::Vector3(1，-1，0));
```

5. 现在创建新的 *ManualObject* 类型并在调用 *begin()* 方法：

```
Ogre::ManualObject*  manual = mSceneMgr->createManualObject("grass");
manual->begin("Examples/GrassBlades"，  RenderOperation::OT_TRIANGLE_LIST);
```

6. 为第一个多边形的顶点添加位置坐标和纹理坐标：

```
manual->position(5.0，  0.0，  0.0);
manual->textureCoord(1，1);
manual->position(-5.0，  10.0，  0.0);
manual->textureCoord(0，0);
manual->position(-5.0，  0.0，  0.0);
manual->textureCoord(0，1);
```

7. 我们也需要第二个三角形拼成一完整的四边形：

```
manual->position(5.0，  0.0，  0.0);
manual->textureCoord(1，1);
manual->position(5.0，  10.0，  0.0);
manual->textureCoord(1，0);
manual->position(-5.0，  10.0，  0.0);
manual->textureCoord(0，0);
```

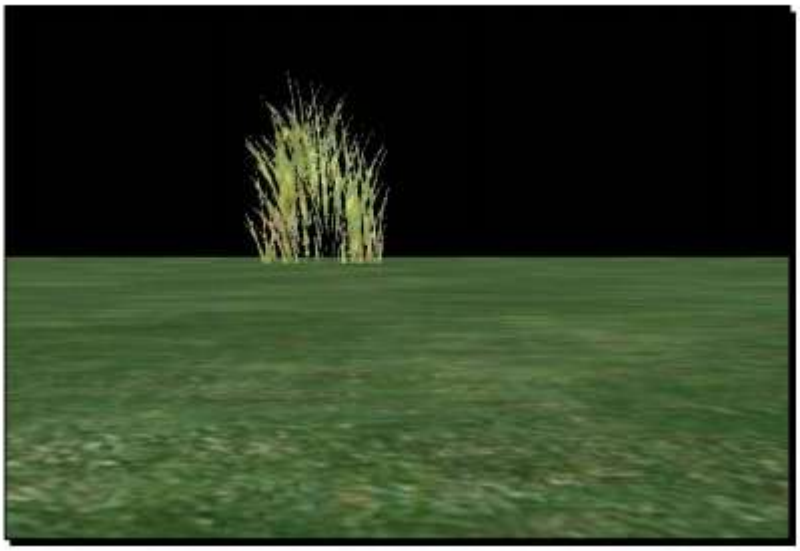
8. 我们已经结束了四边形的定义；让我们通知 manual 对象：

```
manual->end();
```

9. 最后，创建一个新的场景结点并关联 manual object 到它上面：

```
Ogre::SceneNode* grassNode = mSceneMgr->getRootSceneNode()->create
ChildSceneNode("GrassNode2");
grassNode->attachObject(manual);
```

10. 编译运行程序。这个平面上将会有草坪的纹理，并且会有一片草在平面上。



刚刚发生了什么？

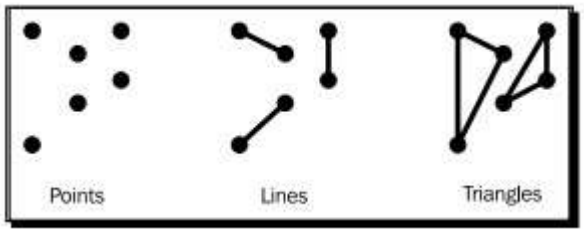
我们用不同的颜色绘制了平面，这次是草绿色；同样，我们创建了四边形并且放置一片草在平面上。第一步和第四步应该理解起来比较简单，我们已经学习过这些。唯一的不同是我们使用了比之前的石头更为适用于应用程序的不同材质。我们将会在下一章好好的讲下关于材质的一些知识

在第五步，我们见到了新的类型。我们创建了一个 ManualObject 对象。

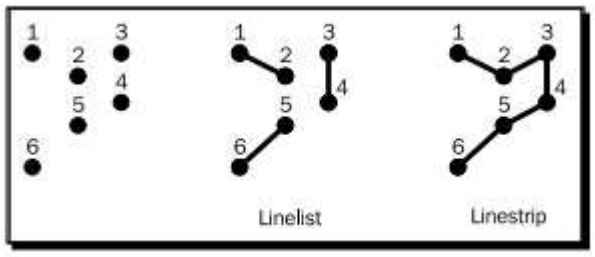
Manual 对象

一个 manual 对象就好像一个新的代码文件。在开始的时候他是空的，但是好多不同的事情可以使用它来创建。使用一个 manual 对象，我们可以创建 3D 模型。为了创建一模型，我们需要给单个顶点来描述三角形。我们已经讨论过我们在 3D 场景中使用的所有对象是由三角形组成。但是对于四边形，我们需要两个三角形，我们将会稍后看到。

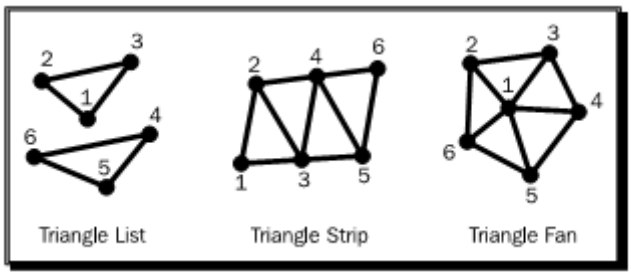
在第五步创建了一个新的空的 manual object 并简单的命名为 grass。我们然后调用了 begin() 方法，这个方法准备接收 manual object 的顶点信息；一个顶点用 3D 坐标来表示。begin()方法需要一个顶点使用的材质的名字，指定一个我们要输出的创建顶点信息。这里有六种方法来指定输出的 manual object 的信息。有三种不同东西可以使用 manual object 创建，即为——点，线和三角形。



点被简单的存储为列。每次我们添加一个新的位置，我们创建一个新的点。这种模式被称为 OT_POINT_LIST。对于线，有两种不同的方法来创建它们。一个比较易懂的方法就是使用第一和第二的点的连线作为第一条线，第三和第四个位置的连线作为第二条线，等等。这被称为 OT_LINE_LIST.另一种方式使用开始两个点作为第一条线，但然后每个新定义的点作为新线的最后一个点并使用最后一个点作为这条线的另一个端点；这是 OT_LINE_STRIP 方式。

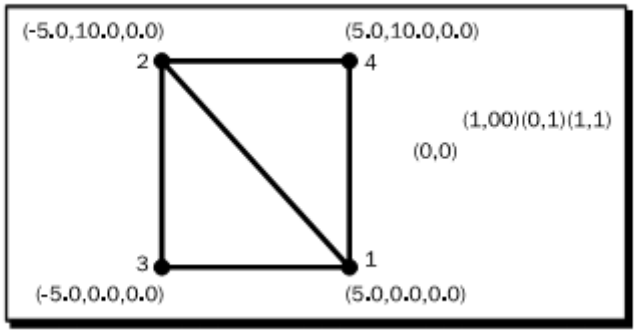


他们可以用三种方式定义三角形。第一种是最简单的方式是 triangle list：前三个点作为一三角形，然后三个作为第二个三角形，如此类推。这被称为 OT_TRIANGLE_LIST。然后我们可以使用前点三个作为第一个三角形并且每个新点使用前两个点定义了的点组成下一个三角形。这被称为 OT_TRIANGLE_STRIP 方式。最后一种方式是使用前三个点作为第一个三角形，然后第一个点，上次最后使用点和新创建的点作为下一个三角形。



我们可以看到，依赖于输入模式，它需要更多的点来描述 3D 图形。用 triangle list 模式，对于每个三角形我们需要三个点。用 strip 或 fan，前三个点为一新三角形，后来每个新的点构成一新三角形。

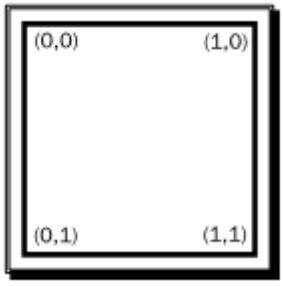
在 begin()函数调用期间，我们定义使用 triangle list 来描述四边形。我们想要四边形宽为 10 个单位并高为 10 个单位.以下四边形有四个点，并且每个点的旁边有坐标标识。



第一个三角形需要点 1，2 和 3。第二个三角形需要点 1，2 和 4。在第六步中用 position()函数定义第一个三角形的各顶点，在第七步调用 position()函数来定义了第二个三角形。你可能注意到在每个 position() 后面调用了 textureCoord() 函数。

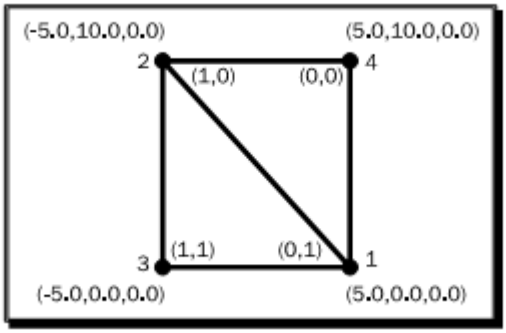
纹理映射

对于 Ogre 3D 可以输出草叶的图像到我们的四边形上，每个顶点除了需要它的位置坐标还需要纹理坐标。纹理坐标由二元组来 (u，v)组成。(u，v) 描述它们在图片中的位置，u 是用于 x 轴，v 是用于 y 轴。(0，0)表示纹理图片的左上角，（1，1）表示纹理图片的右下角。



如果我们使用大于 1 的值，有很多种情况会发生，这取决于材质的设置。如果我们使用 wrap 模式，那么纹理将会重复。如果使用 clamp 模式，每个大于 1 的值将会减到 1.0。如果小于 0 也是用同样的方式—— 坐标会被设为 0。在 mirror 模式中，1 变为了 0，2 变作了 1，在这种情况下反射了纹理，如果它们的值大于 2，原始的图片将会再次使用，翻转之后，又是原图，以此类推。最后的模式定义了边框颜色，在[0，1]之外的所有东西将会被渲染称为边框的颜色。

把纹理坐标应用于原坐标系，我们将会看到下图关于四边形的信息：



让我们看一下第六步和第七步。把代码和上面的图片对比。坐标和纹理坐标的位置将会匹配在一起。

第八步，我们完成了 manual 对象。在第九步，我们创建了一个场景节点来关联我们新创建的对象，这样它就可以被渲染了。

动手试试 —— 使用 manual object

使用 manual object 尝试不同的描述对象的方式。同样，尝试线和点的方式。为使这个过程更加简单，使用 BaseWhiteNoLighting 来代替草地的材质。通过这种材质，你不需要纹理坐标，你只需要使用 position()函数并测试。你创建的所有东西将会被渲染为白色。 】

6.5 给草叶添加体积

我们已经成功渲染了一些草，但是当我们移动摄像机。我们就可以看得清清楚楚那些叶子只是 2D 图形，根本没有什么体积。如果不用我们自己的 3D 模型渲染每片叶子，这个问题不可能简单解决。如果我们做到了，会提升视觉效果，但是这不太容易，会因为大的草地增加渲染的复杂程度，而降低交互的程度。但是有若干技术使这个问题不是太棘手。我们现在可以看一种解决方式。

实践时刻 —— 使用更多的三角形来表现体积

我们将会使用之前的代码并将添加两个新的死表型到我们的草叶上。

1. 在添加过的前两个三角形之后，添加第三个和第四个三角形来创建第二个四边形：

```
//third triangle
   manual->position(2.5，  0.0，  4.3);
   manual->textureCoord(1，1);
```

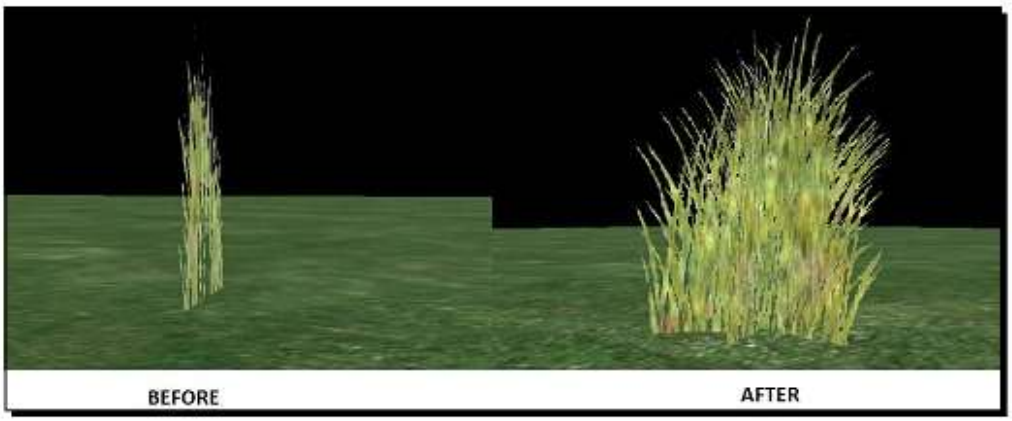
```
manual->position(-2.5， 10.0， -4.3);
manual->textureCoord(0，0);
manual->position(-2.0， 0.0， -4.3);
manual->textureCoord(0，1);

//fourth triangle
manual->position(2.5， 0.0， 4.3);
manual->textureCoord(1，1);
manual->position(2.5， 10.0， 4.3);
manual->textureCoord(1，0);
manual->position(-2.5， 10.0， -4.3);
manual->textureCoord(0，0);
```

2. 添加第五和第六个三角形创建第三个四边形。

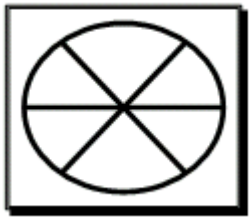
```
//fifth triangle
manual->position(2.5， 0.0， -4.3);
manual->textureCoord(1，1);
manual->position(-2.5， 10.0， 4.3);
manual->textureCoord(0，0);
manual->position(-2.0， 0.0， 4.3);
manual->textureCoord(0，1);
//sixth triangle
manual->position(2.5， 0.0， -4.3);
manual->textureCoord(1，1);
manual->position(2.5， 10.0， -4.3);
manual->textureCoord(1，0);
manual->position(-2.5， 10.0， 4.3);
manual->textureCoord(0，0);
```

3. 编译运行程序，然后在草的周围操控摄像机。在之前的例子中，我们从侧面看只能看到草叶只有一条线的宽度，但现在这种情况不再发生了。

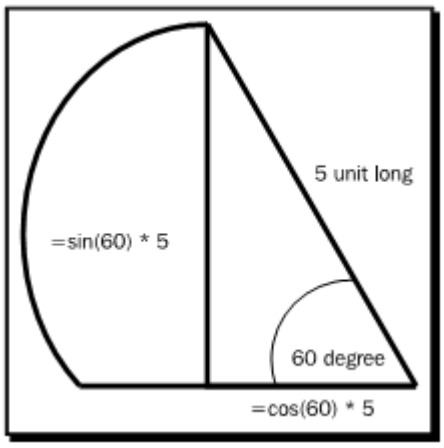


刚刚发生了什么？

我们解决了草叶看起来只像一幅图片投射到四边形上的问题。为了解决这个问题，我们简单的创建了两个新的四边形，位置上有旋转的关系，并把它们相互卡在一起。就好像下面的图示一样：



每个四边形有同样的长，就好像之前的图片一样，我们可以认为它是一个圆分为6份。两个四边形之间有60°的夹角。三个四边形在中心两两相交，这样我们就有6个60度的夹角，最终合成360度。这个图示也回答了之前代码引发的有趣的问题。我们如何计算另外两个四边形各点新坐标？这就是简单的三角形计算。为计算出y的值，我们使用了正弦值，对于x使用了余弦值。我们使用这种方式创建了一个平面并给它渲染一个纹理，使模型更具真实感。这种名为 `billboarding` 的技术，在电子游戏中这种技术被广泛使用。



创建一块草坪

现在我们有一片草，让我们来创建一块完整的草坪吧。

1. 我们需要多个草叶的实例，所以转换 manual object 为 mesh：

```
manual->convertToMesh("BladesOfGrass");
```

2. 我们想要一块草坪包含50 * 50 的草叶实例。所以我们需要两个循环。

```
for(int i=0;i<50;i++)
{
   for(int j=0;j<50;j++)
   {
```

3. 在循环内部，创建一个无名实体和一个无名场景结点：

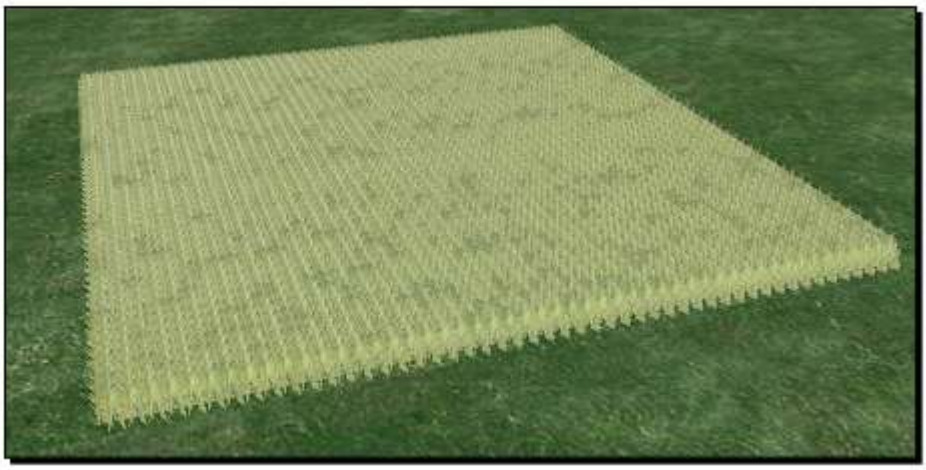
```
Ogre::Entity * ent = mSceneMgr->createEntity("BladesOfGrass");
Ogre::SceneNode* node =
mSceneMgr->getRootSceneNode()->createChildSceneNode(Ogre::Vector3(i*3，-10，j*3));
node->attachObject(ent);
```

4. 不要忘记关闭循环：

```
    }
```

```
}
```

5. 编译运行程序，你将会看到一块草坪。看你的电脑的配置，这块草坪可能非常慢才会渲染完成。



刚刚发生了什么？

在第一步，我们使用了一个新的 manual object 的成员函数。这个函数把manual object转变一个mesh，这样我们就可以使用场景管理器的createEntity()函数来创建mesh的实例。为能使用新的实体，我们需要准备一个稍后在createEntity()函数中使用的参数名。这次，我们使用BladesOfGrass作为描述mesh的名字。我们想要若干草的实例，所以我们在第二步创建了两个循环，每个50次。在第三步中添加了循环体。在循环体中，我们首先使用刚创建的参数名新建了一个实体。有心的读者可能注意到了我们没有用两个参数传入createEntity()函数，即一个实体类型和一个想要创建的参数名。这次，我们只给了实体的类型作为实参，而非名字。但是否是因为每个实体都需要一个独一无二的名字，所以每次必须给它一个名字呢？这个想法是正确的，我们调用的函数仅是一个辅助函数，它只需要一个实体类型的名字，因为它会自己生成一个唯一的名字，然后调用我们经常用的函数。这为我们节省了为循环附加如BladesOfGrassEntity一般的变量名字的而带来的麻烦。我们使用同类型的函数来创建场景结点。

探索名称生成方案

现在，让我们快速浏览一下Ogre 3D 为我们生成的名字。

1. 在循环体的结尾，添加下面的输出语句：

```
std::cout << node->getName() << "::" << ent->getName() <<
```

```
std::endl;
```

2. 编译运行程序;将会有打印出来一长列的名字。更准确的说，是2500行，因为循环重复了50*50次。下面是最后几行的名字：

Unnamed_2488::Ogre/MO2487

Unnamed_2489::Ogre/MO2488

Unnamed_2490::Ogre/MO2489

Unnamed_2491::Ogre/MO2490

Unnamed_2492::Ogre/MO2491

Unnamed_2493::Ogre/MO2492

Unnamed_2494::Ogre/MO2493

Unnamed_2495::Ogre/MO2494

Unnamed_2496::Ogre/MO2495

Unnamed_2497::Ogre/MO2496

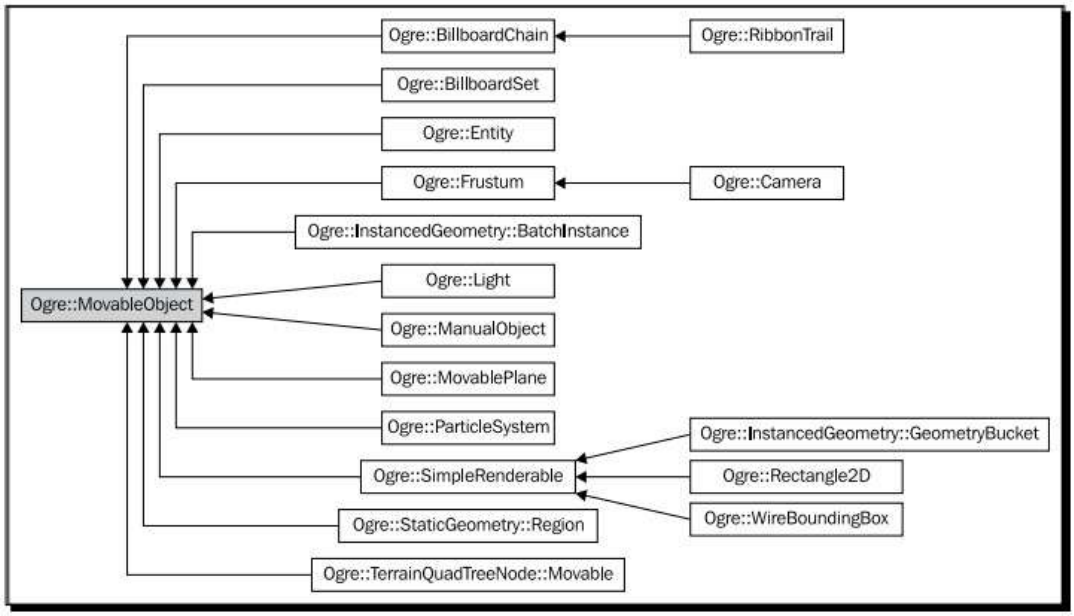
Unnamed_2498::Ogre/MO2497

Unnamed_2499::Ogre/MO2498

Unnamed_2500::Ogre/MO2499

刚刚发生了什么？

我们刚刚打印出场景结点的名字，并且当我们不赋一个名称作为参数，创建的实体成功的识别了Ogre 3D自动生成的名称。我们看到使用的场景结点使用了这种格式：Unnamed_Nr，Nr是一个计数器，每次我们创建一个无命名的场景时，它便会增加。实体使用了一个相似的方案，但使用的是MO格式；MO是movable object的简写。 moveable object 在Ogre 3D中，可作为多种不同类的基类。可借助场景结点移动的所有东西都是继承自一个movable object。如实体和光源都是继承自 Movable object 类，但还有更多的继承自movable object。下面是一幅来自于Ogre 3D 文档的图片显示了从MovableObject继承的所有类。



资料来源：http://www.ogre3d.org/docs/api/html/classOgre_1_1MovableObject.html

我们看到甚至摄像机是也是一个movable object；这是很必要，否则我们不能把它关联到场景结点上。只有MovableObject的子类可以可以被关联到场景结点上。名字不同的实体被关联到结点上，并且这步是用下面的代码来实现添加实体到无名结点上的。

```
mSceneMgr->getRootSceneNode()->createChildSceneNode()->attachObject(ent);
```

6.6 静态几何

我们创建了一块草坪，但是程序可能因你电脑的原因，运行的相当慢。你可以使用Ogre 3D中一个称为 *StaticGeometry* 的类来使程序运行的更快。

实践时刻 —— 使用静态几何

我们将会修改上个例子中的代码，使程序渲染的更快：

1. 删除打印语句；我们下面不再需要它了。

2. 现在回到 *manual object* 的话题，删除所有添加相同点的 *position()* 函数的调用。对于每个四边形，应有四或六个点。下面是删除多余实体后的代码：

```
manual->position(5.0， 0.0， 0.0);
manual->textureCoord(1，1);
```

```
manual->position(-5.0， 10.0， 0.0);
manual->textureCoord(0，0);
manual->position(-5.0， 0.0， 0.0);
manual->textureCoord(0，1);
manual->position(5.0， 10.0， 0.0);
manual->textureCoord(1，0);
manual->position(2.5， 0.0， 4.3);
manual->textureCoord(1，1);
manual->position(-2.5， 10.0， -4.3);
manual->textureCoord(0，0);
manual->position(-2.0， 0.0， -4.3);
manual->textureCoord(0，1);
manual->position(2.5， 10.0， 4.3);
manual->textureCoord(1，0);
manual->position(2.5， 0.0， -4.3);
manual->textureCoord(1，1);
manual->position(-2.5， 10.0， 4.3);
manual->textureCoord(0，0);
manual->position(-2.0， 0.0， 4.3);
manual->textureCoord(0，1);
manual->position(2.5， 10.0， -4.3);
manual->textureCoord(1，0);
```

3. 现在我们使用称为索取的方法来描述想要创建的三角形。第一个三角形使用了头三个点，第二个三角形使用了第一，第二和第四个点。记住，和计算机中别的概念是一样，点是从 0 开始计数的：

```
manual->index(0);
manual->index(1);
manual->index(2);
manual->index(0);
manual->index(3);
manual->index(1);
```

4. 用同样的方式添加另外两个四边形：

```
manual->index(4);
manual->index(5);
manual->index(6);
manual->index(4);
manual->index(7);
manual->index(5);
manual->index(8);
manual->index(9);
manual->index(10);
manual->index(8);
manual->index(11);
manual->index(9);
```

5. 现在让SceneManager 创建一个新的静态实例：

```
Ogre::StaticGeometry* field = mSceneMgr->createStaticGeometry("FieldOfGrass");
```

6. 现在在for循环中，创建草的实体。然而，这次添加它到静态几何的实例而不是场景结点：

```
for(int i=0;i<50;i++)
{
   for(int j=0;j<50;j++)
   {
     Ogre::Entity * ent = mSceneMgr->createEntity("BladesOfGrass");
     field->addEntity(ent，Ogre::Vector3(i*3，-10，j*3));
   }
}
```

7. 调用build函数，来结束静态几何：

```
field->build();
```

8. 编译运行程序，你将会看到同样的一片草地，但是这次，程序将会运行的快很多。

刚刚发生了什么？

我们创建了我们之前一样的场景，但是这次运行的更快了。为什么呢？唯一的原因是静态几何运行的更快。但静态几何和我们平时用的方法有什么不同？

渲染线管

每次我们渲染一个场景，Ogre 3D和显卡都需要完成一些步骤。我们已经讲过一些，但是不是全部。我们讲过的一步是拣选；现在我们讨论一些我们没有遇到过的步骤。

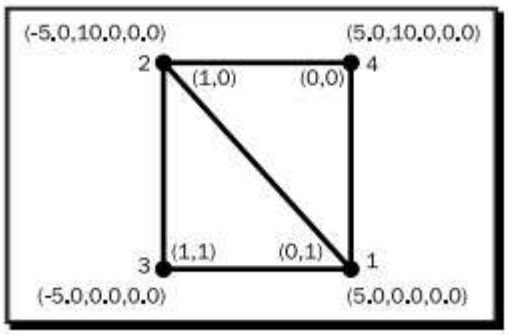
我们知道对象可以在不同的空间中，如局部空间或世界空间。我们也知道对渲染一个对象，我们需要把它们从局部空间变为世界空间。从局部空间到世界空间的变换是简单数学运算的组和，Ogre 3D 需要每帧计算每个草实体的世界坐标。每帧需要很多的操作，但是更糟的是每个草实体被独立地送给GPU渲染。这花费了很多的时间，而且这就是为什么程序运行的很慢。

我们可以使用静态几何来解决这个问题；在第五步，我们使用场景管理器创建了一个静态几何的实例。然而，在循环的内部，我们添加创建的实体，而不是像使用过的关联到场景结点上的那种方式。这里，我们直接添加它到静态几何，并给出想要实体的位置作为第二参数。

在我们添加完实体到静态几何实例之后，我们需要调用build()函数。这个函数代入我们添加的所有实体，并计算出世界坐标，甚至做更多的事情。我们只需要使用索引列表添加模型，因为静态几何尝试使用同样的材质或索引和模型本身联系起来以更大的优化程序。我们付出的代价就是不能移动添加到静态几何的实体。在草地的那种情况下，这并不是什么损失；草是保持不动的。通常，静态几何被用于场景中不动的东西，因为它提供了一个几乎没有缺点的加速效果。但一个缺点就是当我们在场景中有大量的静态实例，当一部分静态几何在视锥范围之内，拣选就变得不太有效了，因为每个对象都必须渲染。

索引

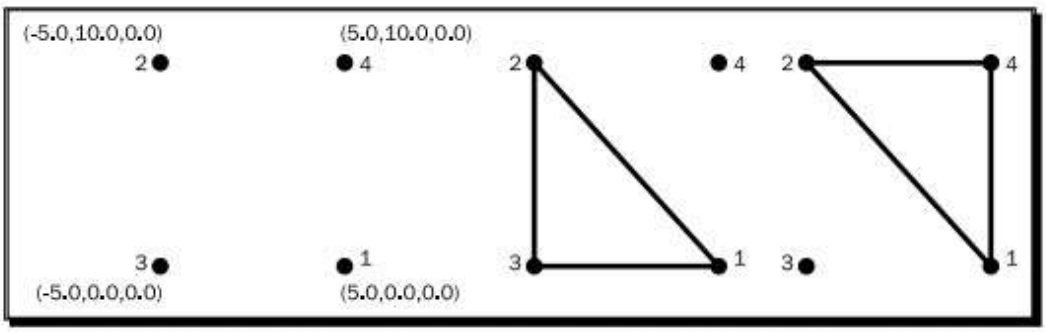
我们发现只可添加已给静态几何实例加过索引的实体。但首先我们没有讨论过什么是索取。为理解这个概念，让我们回到四边形的话题。



四边形有四个点，是由两个三角形组成并定义的。当我们研究下过去使用创建四边形的代码，注意到我们直接使用添加6个点的方式来代替四个点两点被重复添加的方式。

```
//First triangle
manual->position(5.0， 0.0， 0.0);
manual->textureCoord(1，1);
manual->position(-5.0， 10.0， 0.0);
manual->textureCoord(0，0);
manual->position(-5.0， 0.0， 0.0);
manual->textureCoord(0，1);
//Second triangle
manual->position(5.0， 0.0， 0.0);
manual->textureCoord(1，1);
manual->position(5.0， 10.0， 0.0);
manual->textureCoord(1，0);
manual->position(-5.0， 10.0， 0.0);
manual->textureCoord(0，0);
```

在图中点1 和点2，被添加了两次，因为它们被两个三角形所使用。一种防止信息重复的方式就是使用two-step system来描述三角形。首先，我们创建了一个想要使用的点列。第二，我们创建在点外定义了我们想要生成的三角形的列。



这里，我们定义了四个点，然后告诉三角形使用点 1，2和3，并且第二个三角形使用点2，2，和4。这节省了我们添加一些点两次，或更多的时候，大于两次。这看起来只有小小的不同，但是当我们有几千个模型的时候，这就会产生重大的影响了。静态几何要求我们只使用添加过索引的实体，因为用这种方法，静态几何可以创建一个有所有点(也就是顶点)的列表和有所有索引的列表。如果我们使用同样的点添加实体，静态几何只需要添加一些索引而不是新的点。对于复杂的处理来说，这将会是一个很大的空间节省。

6.7 小结

我们已经改变了目前的场景管理器，创建我们自己的草坪，用静态几何加速程序。

具体的说，我们学习了：

1. 什么是ManualObject。
2. 为什么我们为3D模型使用索引。
3. 如何和何时使用静态几何。

在这章我们已经使用过材质。在下一章，我们将会创建自己的材质。

第七章 Ogre 3D 与材质

没有材质，我们不能给场景添加细节，而这章将对材质的广泛的应用给以介绍。

材质是一个很重要的概念，而且对产生漂亮的场景来说，非常有必要去理解。材质对尚在进行的研究来说也是个有趣的话题。

这一章，我们将会：

1. 学习如何创建我们自己的材质。
2. 应用纹理到我们的四边形。
3. 更好的理解渲染线管是如何工作的。
4. 使用着色器来创建其他技术很难实现的效果。

现在，让我们开始吧….

7.1 创建一个白色的四边形

在之前的章节中，我们用代码创建了自己的 3D 模型。现在，我们将会使用这种办法来创建一可做实验的四边形例子。

实践时刻 —— 创建一个四边形

我们将以一个空的程序作为开始，并插入创建四边形的代码到 *createScene()* 函数：

1. 开始创建 manual object：

```
Ogre::ManualObject*    manual = mSceneMgr->createManualObject("Quad");
manual->begin("BaseWhiteNoLighting",    RenderOperation::OT_TRIANGLE_LIST);
```

2. 为四边形创建点：

```
manual->position(5.0,    0.0,    0.0);
manual->textureCoord(0,  1);
manual->position(-5.0,    10.0,    0.0);
manual->textureCoord(1,  0);
manual->position(-5.0,    0.0,    0.0);
```

```
manual->textureCoord(1，1);
manual->position(5.0，  10.0，  0.0);
manual->textureCoord(0，0);
```

3. 使用索引来描述四边形：

```
manual->index(0);
manual->index(1);
manual->index(2);
manual->index(0);
manual->index(3);
manual->index(1);
```

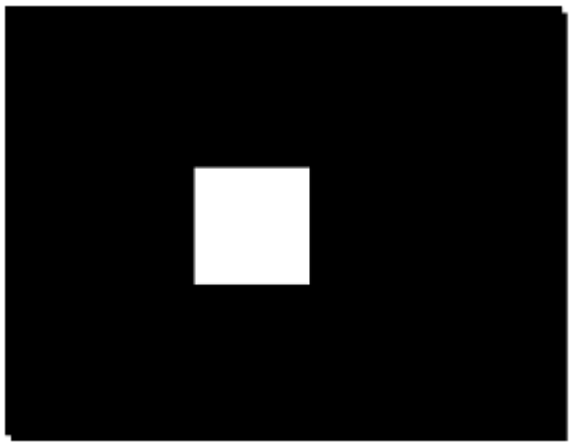
4. 完成 manual object 并转换它们为 mesh：

```
manual->end();
manual->convertToMesh("Quad");
```

5. 创建实体的实例，并使用场景结点到场景：

```
Ogre::Entity * ent = mSceneMgr->createEntity("Quad");
Ogre::SceneNode* node = mSceneMgr->
                                    getRootSceneNode()->createChildSceneNode("Node1");
node->attachObject(ent);
```

6. 编译运行程序。你将会看到一个白色的四边形。



刚刚创建了什么？

我们使用之前章节的知识创建了一个四边形并与它关联一个白色的材质。下一步我们要做的是就是创建自己的材质。

7.2 创建我们自己的材质

总是渲染所有的东西为白色并不是多么有趣的事情，所以让我们创建我们自己的第一个材质。

实践时刻 —— 创建材质

现在，我们将会使用创建的白色四边形来创建自己的材质。

1. 在程序中把材质的名称 BaseWhiteNoLighting 改变为 MyMaterial1：

```
manual->begin("MyMaterial1",  RenderOperation::OT_TRIANGLE_LIST);
```

2. 在 Ogre 3D SDK 的 *media\materials\scripts* 文件夹创建一个新的文件名字为 *Ogre3DBeginnersGuide.material*。

3. 写入如下代码到刚才的文件中：

```
material MyMaterial1
{
   technique
```

```
    {
        pass
        {
            texture_unit
            {
                texture leaf.png
            }
        }
    }
}
```

4. 编译运行程序。你将会看到在白色的四边形上有一片植物。

刚刚发生了什么？

我们创建了自己的第一个材质文件。在 Ogre 3D 中，材质可以在 material 文件中定义。为找到我们的材质文件，我们需要在 resources.cfg 中给出其材质文件的目录列表。我们也直接在代码中使用 ResourceManager 给出它们在文件中的路径，就好像我们在之前章节中介绍的加载地址的方式相似。为使用在材质文件中定义的材质，我们需要在开始调用 manual object 中使用材质的名称。

真正有趣的部分是材质文件

材质

每个材质是以材质的关键字作为开始，也就是材质的名字，然后就是”{ “。在材质的结尾使用了”}”，这种技术目前对你来说应十分熟悉了。每个材质由一种或多种技术(technique)组成；每种技术描述了实现预先效果的方式。因为不同的显卡有着不同的特性，我们可以定义多种技术，Ogre 3D 从开始到最后的遍历文件一遍，选择第一种用户显卡支持的技术。在一种技术里面，你可以有多种 pass。一个 pass 是独立几何的渲染。对于我们将要创建的大部分材质，只需要一个 pass。然而，对于创建的更复杂的一些材质需要两个或三个的 pass，所以 Ogre 3D 允许我们每种技术定义几种 pass。在这个 pass 里，我们只定义了一个纹理单位。一个纹理单位定义了一个纹理和它的属性。这次我们定义的唯一属性就是纹理可以被使用。我们使用 leaf.png 作为使用的纹理图像。这个纹理是 Ogre SDK 自带的，并且是 resources.cfg 可以所以到得文件，这样我们可以不遗余力的使用它了。

动手试试 —— 创建另一个材质

使用 Water02.jpg 创建一个称为 MyMaterial2 的新纹理

7.3 纹理坐标

在之前的章节中，我们讨论了当纹理坐标超出 0 到 1 的范围时，各个方式的使用策略。现在，让我们创建一些材质在看看他们的实际效果。

实践时刻 —— 准备我们自己的四边形

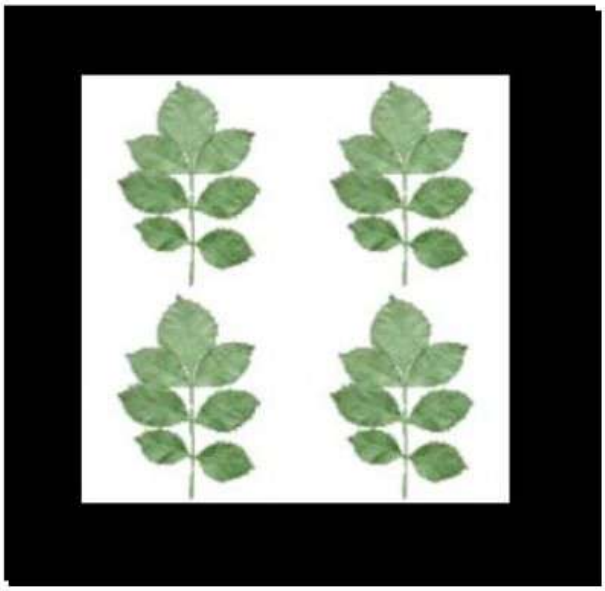
我们将会使用之前有叶子纹理的四边形那个例子：

1. 把四边形的纹理坐标范围从[0，1]变为[0，2]。这个四边形的代码将会如下所示：

```
manual->position(5.0， 0.0， 0.0);
manual->textureCoord(0，2);
manual->position(-5.0， 10.0， 0.0);
manual->textureCoord(2，0);
manual->position(-5.0， 0.0， 0.0);
manual->textureCoord(2，2);
manual->position(5.0， 10.0， 0.0);
manual->textureCoord(0，0);
```

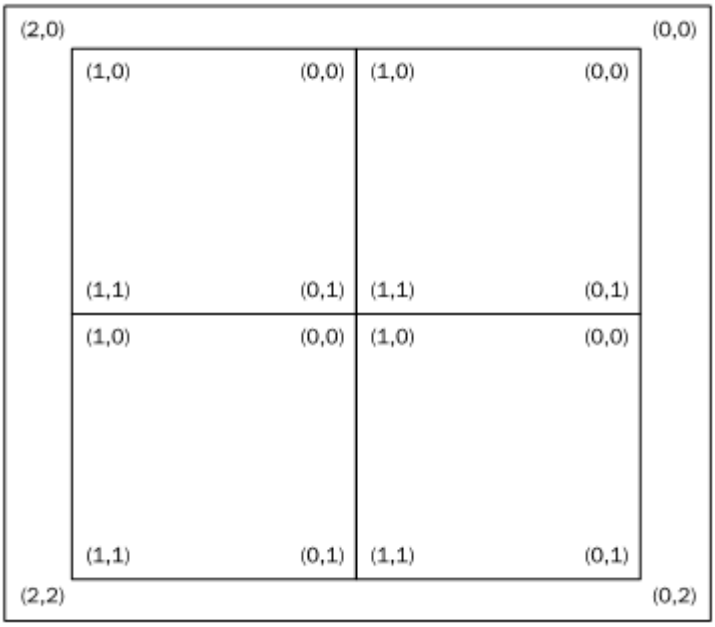
2. 现在编译运行程序。如往常一般，我们将会看到在四边形上有一片叶子的纹理，但

是这次我们会看到四个纹理。



刚刚发生了什么？

我们简单的改变了我们的四边形，使得纹理坐标的范围为[0，2]。这意味着 Ogre 3D 需要使用它本身的策略以渲染大于 1 的纹理坐标。默认的模式为重复寻址模式(wrap)。这意味着每个超过 1 的值将会覆盖为 0 和 1 范围之间。下面的图示显示了纹理坐标是如何被覆盖的。在边角外部，我们可以看到纹理坐标的原点在边角的内部，我们看到了覆盖之后的纹理坐标的值。同样，为了更好的理解，我们可以看到四个重复的纹理，它们显式的坐标表示。



我们已经看到默认的纹理 wrapping 模式是如何覆盖我们的纹理的。我们的平面纹理显示的效果很好，但是它并没有显示这个技术的用处。让我们使用另一个纹理来看下 wrapping 模式下的好处。以另一个纹理使用 wrapping 模式。

实践时刻 —— 添加一个岩石纹理

对于这个例子，我们将会使用另一个纹理。否则，我们将不会看到这个纹理模式的效果。】

1. 创建同上一材质相似的新材质，但是这个把材质变为：*terr_rock6.jpg* ：

```
material MyMaterial3
{
    technique
    {
        pass
        {
            texture_unit
            {
                texture terr_rock6.jpg
            }
        }
    }
}
```

2. 从 *MyMaterial1* 改变我们使用的材质为 *MyMaterial3*：

```
manual->begin("MyMaterial3",  RenderOperation::OT_TRIANGLE_LIST)
```

3. 编译运行程序。你将会四边形会有岩石的纹理：

刚刚发生了什么？

这次，这个四边形看起来好像是一个单独的纹理。我们看不到什么明显的重复，效果如平面纹理一般。原因如我们所知的一样，这是 wrapping 模式重复的结果。在一块纹理的最左边，纹理又以其右边重新开始重复，纹理下方的重复同理。这种纹理被称为无缝纹理。我们使用的纹理使左边，右边和上，下边完美的拼接起来。如果不是这种情况，我们就可以看到纹理明显重复的痕迹。

7.4 使用另一个纹理模式

我们已经看到 *wrapping* 模式的效果和用处。现在，让我们看一下另一种称为 clamping 的纹理模型】

实践时刻 —— 添加一个岩石纹理

我们将会使用相同的项目，并仅创建一个新的纹理：

1. 创建一个新的成为 MyMaterial4 的纹理，纹理的内部和之前的纹理一样：

```
material MyMaterial4
```

```
{
    technique
    {
        pass
        {
            texture_unit
            {
                texture terr_rock6.jpg
            }
        }
    }
}
```

2. 在纹理单位段中，添加一行代码告诉 Ogre 3D 使用 clamp 模式：

```
tex_address_mode clamp
```

3. 把我们的四边形材质的名字从 *MyMaterial3* 改变为 *MyMaterial4*：

```
manual->begin("MyMaterial4"， RenderOperation::OT_TRIANGLE_LIST);
```

4. 编译运行程序。你应会看到四边形纹理左上的石头纹理。四边形的另外三块是由不同颜色的线组成的。

刚刚发生了什么？

我们改变纹理模式为 clamp。这个模式使用纹理的边缘像素来填充所有大于 1 的纹理坐标、在实践中，这意味着一个图片的边缘在模型上得到了拉伸；我们可以之前的图片中看到这种效果。

7.5 使用 mirror 模式

让进入下一个我们可使用的纹理模式。

实践时刻 —— 使用 mirror 模式

1. 使用之前的材质作为模板，创建一个新的称为 MyMaterial5 的材质。

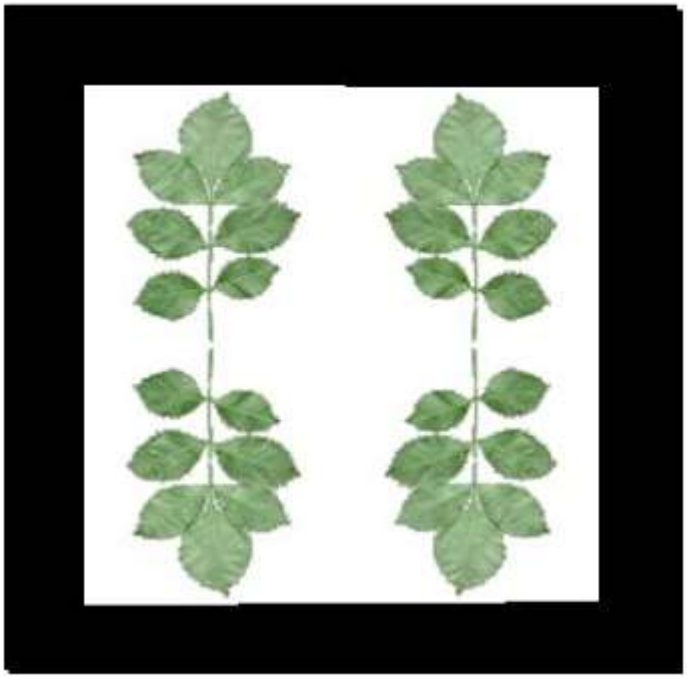
2. 改变材质模式为 mirror：

```
tex_address_mode mirror
```

3. 改变纹理为我们之前使用过的叶子纹理：

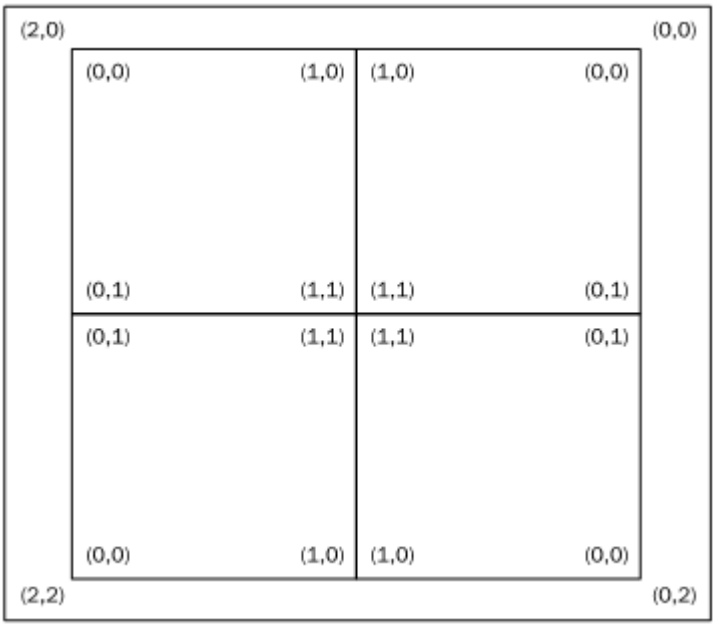
```
texture leaf.png
```

4. 编译运行程序，你将会看到看到叶子被反射了四次



刚刚发生了什么？

我们又一次的改变了纹理模式——这次是 mirroring。当用于渲染一块像石墙那样的大面积区域时，使用 mirror 模式就显得比较简单而有效了。每次纹理坐标值大于 1 时，纹理就得到翻转，然后就如使用 wrap 模式一般了。我们可以在下面的图示中看到 mirror 模式显示的效果。



7.6 使用 border 模式

最后一个需要尝试的模式，即为，border 模式。

实践时刻 —— 使用 border 模式

1. 创建一个成为 MyMaterial6 的材质，就如前面五次一样，也是基于之前已使用过的材质。

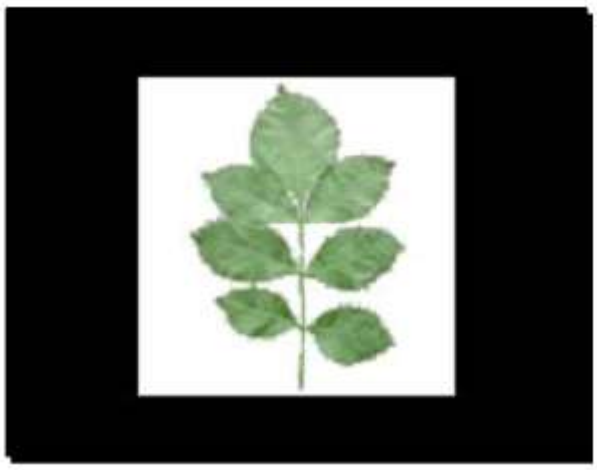
2. 改变纹理模式为 border 模式：

```
tex_address_mode border
```

3. 同样记住改变在代码在我们使用的材质名称：

```
manual->begin("MyMaterial6"， RenderOperation::OT_TRIANGLE_LIST);
```

4. 编译运行程序。令人惊讶的是，这次我们只看到一片叶子。



刚刚发生了什么？

别的叶子都去哪里了？border 模式并不是如 mirro 模设计或 wrap 模式一般创建了纹理的多份拷贝。当纹理坐标大于 1 的时候，这个模式画任何东西为边框颜色——默认的颜色显然是黑色的，黑色的 RGB 颜色值为(0，0，0)。

7.7 改变边框的颜色

如果我们只使用黑色作为边框颜色，这个特性就没有什么用处了。让我们看下如果改变边框的颜色。

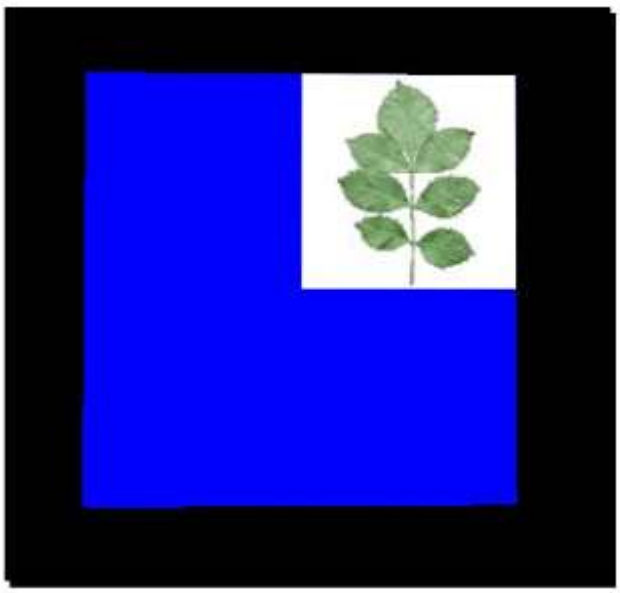
实践时刻 —— 改变边框的颜色

1. 复制最后一个材质，并命名为 *MyMaterial7*。

2. 在设置完材质的模式之后，添加下面的代码以设置边框的颜色为蓝色：

```
tex_border_colour    0.0 0.0 1.0
```

3. 编译运行程序。这次，我们也只看到一片叶子的纹理，但是四边形的剩余部分为蓝色。



刚刚发生了什么？

我们从黑色改变边框的颜色为蓝色。类似的，我们可以使用任意颜色作为边框颜色，边框颜色可以用 RGB 值来表示。这个纹理模式可以用于放置 logo 到像赛车这样的对象。我们只需要设置边框的颜色为车的颜色，然后添加材质。如果纹理坐标存在错误或不准确，它们就不会显示出来不对的地方，因为车和边框颜色都是一样的。

简单测试 —— 纹理模式

四种纹理模式——wrap， clamp， mirror， 和 border 之间的有什么不同？

a. 处于 0 和 1 之间的纹理坐标是如何使用的？
b. 高于 1 或小于 0 的场景坐标是如何处理的？
c. 纹理的颜色是如何渲染的？

动手试试 —— 使用纹理模式

尝试使用大于 2 或小于 0 的纹理坐标。

7.8 滚动一个纹理

我们已经看到过几种纹理模式，但是这只是一材质文件的一个属性。现在，我们将会使用另一种相当有用的属性。

实践时刻 —— 准备滚动一个纹理

这次，我们将会改变我们的四边形来观察新材质的效果：

1. 改变使用过的材质为 MyMaterial8 并同时改变纹理从 2 到 0.2：

```
manual->begin("MyMaterial8"， RenderOperation::OT_TRIANGLE_LIST);
manual->position(5.0， 0.0， 0.0);
manual->textureCoord(0.0，0.2);
manual->position(-5.0， 10.0， 0.0);
manual->textureCoord(0.2，0.0);
manual->position(-5.0， 0.0， 0.0);
manual->textureCoord(0.2，0.2);
manual->position(5.0， 10.0， 0.0);
manual->textureCoord(0.0，0.0);
```

2. 现在创建在材质文件中创建新的材质 MyMaterial8。这次，我们不需要任何纹理模式；仅使用纹理 terr_rock6.jpg 就可以了：

```
material MyMaterial8
{
    technique
    {
```

```
            pass
            {
                texture_unit
                {
                    texture terr_rock6.jpg
                }
            }
        }
}
```

3. 编译运行程序。你将会看到我们之前看到的石头纹理的一部分.

刚刚发生了什么？

我们只能看到纹理的一部分，那是因为我们的四边形的纹理坐标最大到 0.2；这意味着五分之四的纹理不能渲染到我们的四边形。这这次“实践时刻”中发生的所有事理解起来应很简单，因为它只是我们目前这章学到东西的重复。如果你不太理解，请重新复习下这章。

实践时刻 —— 滚动一个纹理

现在我们准备好四边形，开始滚动纹理：

1. 添加下面一行代码到材质文件的纹理设置代码段以滚动纹理：

```
scroll 0.8 0.8
```

2. 编译运行程序。这次，你应看到纹理的不同部分。

刚刚发生了什么？

滚动的属性改变了纹理坐标的给定偏移量。下面的图示显示了滚动的效果。右上角是我们所渲染的纹理的第一部分，左下角是滚动属性应用到渲染纹理的一部分。 】

我们可以在不需要修改模型本身 UV 坐标系的情况下，来改变纹理坐标的属性。

7.9 动画滚动

能够在材质中滚动纹理并不十分惊人。但是它较完整渲染模型来说，可以帮助我们节省一些渲染时间。让我们添加一些动态的滚动。

实践时刻 —— 添加动画滚动

我们也使纹理的滚动时动态的。让我们开始实现吧。

1. 创建一个新的材质并改变滚动的属性来使动画滚动：

```
scroll_anim 0.01 0.01
```

2. 记住同样改变使用的 manual object 的材质;否则，你将看不到任何的改变。

3. 编译运行程序。当仔细观察的时，你应发纹理从右上移动到左下角。我不能显示这张图片，因为书上的图片是不能动的(可能未来有可能实现下)。

刚刚发生了什么？

我们使用了另一个属性使纹理滚动。除了名字，这个属性同滚动属性的名字几乎相同，但虽不起眼，但重要之区别处在于我们现在设置的偏移量是每秒。

还有更多我们可操纵的纹理属性。一个完整的列表可在 http://www.ogre3d.org/docs/manual/manual_17.html#SEC9 找到。

7.10 继承材质

在我们接触像着色器般的复杂话题之前，我们将会尝试从继承材质

实践时刻 —— 从材质中继承

我们将会创建两个新的材质。我们同样会改变四边形的定义：

1. 对于这个例子，我们需要一个四边形来显示一个纹理。改变四边形的定义使用 0 到 1 之间的纹理坐标，并记住改变使用的材质为接下来创建的 *MyMaterial11*：

```
manual->begin("MyMaterial11",  RenderOperation::OT_TRIANGLE_LIST);
manual->position(5.0,  0.0,  0.0);
manual->textureCoord(0.0,  1.0);
manual->position(-5.0,  10.0,  0.0);
manual->textureCoord(1.0,  0.0);
manual->position(-5.0,  0.0,  0.0);
```

```
manual->textureCoord(1.0，1.0);
manual->position(5.0， 10.0， 0.0);
manual->textureCoord(0.0，0.0);

manual->index(0);
manual->index(1);
manual->index(2);
manual->index(0);
manual->index(3);
manual->index(1);
manual->end();
```

2. 新的材质将会使用岩石材质，并使用 *rotate_anim* 的属性，材质将会以给定的速度旋转。但是最重要的事情是命名纹理单元 *texture1*：

```
material MyMaterial11
{
    technique
    {
        pass
        {
            texture_unit texture1
            {
                texture terr_rock6.jpg
              rotate_anim 0.1
            }
        }
    }
}
```

3. 现在创建第二个四边形，并沿 X 轴移动 15 个单位，这样它就不会与第一个四边形相交。同样使用 *setMaterialName()* 函数来改变被实体使用的材质为 *MyMaterial12*：

```
ent = mSceneMgr->createEntity("Quad");
```

```
ent->setMaterialName("MyMaterial12");
node = mSceneMgr->getRootSceneNode()->createChildSceneNode("Node2"，Ogre::Vect
or3(15，0，0));
node->attachObject(ent);
```

4. 最后要做的是创建 *MyMaterial12*。我们将会从 *MyMaterial11* 继承并设置纹理别名(texture alias)为另一个可使用的纹理：

```
material MyMaterial12 : MyMaterial11
{
   set_texture_alias texture1 Water02.jpg
}
```

5. 编译运行程序，然后你将会看到两个一直旋转的四边形——一个是岩石纹理，另一个是水纹纹理。

刚刚发生了什么？

我们创建了两个四边形，每个都有其自己的材质。在第一步和第二步中，仅修改四边形的纹理坐标，使其在[0，1]的范围之内。在第二步中，我们创建了四边形的材质并使用新属性 *rotate_anim x* ，这个属性每秒旋转纹理 x 轴的方向——没有什么复杂。同样我们给纹理单位命名为 *texture1*；稍后我们将使用这个名字。在第三步中，我们创建另一个四边形的实例并使用 *setMaterialName()* 函数改变被实体使用的材质。第四步是重要的一步。这里使用

继承创建了一新材质，继承的概念我们应比较熟悉，和 C++中的语法是一致的，*NewName : ParentName* 。在这种情况下，*MyMaterial12* 继承 *MyMaterial11*。然后我们使用了 *set_texture_alias* 属性以绑定纹理 *Water02.jpg* 到纹理单元 *texture1*。这种情况下，我们用 *Water02.jpg* 代替 *terr_rock6.jpg*。因为这是创建新材质代码中唯一的改变，我们现在可以在这里停止了。

纹理别名的使用使我们可以创建很多材质，我们不需要对每个材质从顶层从头写起，只需要改变纹理之中的不同就可以了，而且我们都知道在尽可能的情况下尽量避免重复。

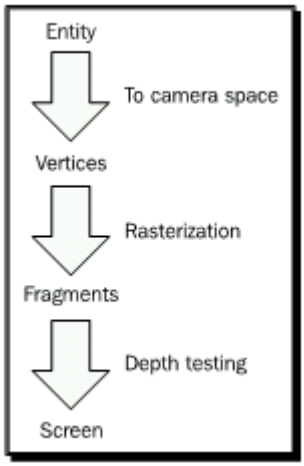
我们已经了解了关于材质很多的事情，但是我们还需要做很多东西。我们所学已经涵盖了纹理的基础概念，在文档的帮助下，现在应可以理解可在材质中使用的别的大部分属性了。请参考这里 http://www.ogre3d.org/docs/manual/manual_14.html#SEC23 。我们将会进一步深入，并学习如何用所谓的 shaders 编写我们自己的显卡程序。

7.11 固定线管和着色器

在这一章，我们一直使用一种称为固定线管的技术。这是一种可在显卡上渲染出很好效果图像的渲染线管技术。正如名字中固定所暗示的那样，并没有为操作者提供太大的操控固定线管的自由。我们可以使用材质文件微调一些参数，但是这没什么意思。这时着色器可以帮助填补这个空白。着色器是可被显卡加载的小程序，而且函数可以作为渲染处理的一部分。这些着色器可被看做以 C 语言风格，书写虽小，但是很给力的函数程序集。用着色器，我们可以几乎完全控制场景的渲染，而且可以添加仅用固定线管实现不了的新特效。

渲染线管

为理解着色器，我们需要对渲染的整个过程原理有个首先的理解。当渲染的时候，我们模型的每个顶点坐标从局部空间变为摄像机空间坐标，然后每个三角形光栅化。这意味着，显卡会计算如何在图像中代替模型。这些图片被称为片段。每个片段随后被处理和操作。我们可以应用纹理的一个特殊部分到这个片段来为模型贴图或者当以一种颜色渲染模型时，我们需要简单赋给它一个颜色。在这种处理过后，显卡测试片段是否被另一更接近摄像机的片段所覆盖，或者它是否是接近摄像机的最近片段。如果是这种情况，这个片段就可以显示在屏幕上。在较新的硬件中，这一步可以在处理片段之前发生。如果在最后的结果中，大部分的场景是不可见的，这种方法可以节省很多计算时间。下面一个简单的图示显示出了线管的步骤：



对于几乎所有的新一代显卡，新的着色器类型就会被引入。开始的时候是 顶点，像素/片段着色器。顶点着色器的任务是变换顶点到摄像机空间，如果需要的话，可以用任何方式来修改，就像在 GPU 上做完整的动画。像素和片段着色获得器光栅片段，而且一种别的方式应用纹理或操作它们，比如，对于有像素精度的光源模型。同样存在别的着色器方案，比如几何着色器，但是我们将不会再这本书中做讨论，因为他们是比较新的技术，并不是广泛支持的，而且也超出了这本书的讲述范围。

实践时刻 —— 我们第一个 shader 程序

让我们开始写我们第一个顶点和片段着色器

1. 在我们的程序中，我们仅需要改变使用过的材质。改变它为 MyMaterial13。同样移除第二个四边形：

```
manual->begin("MyMaterial13",  RenderOperation::OT_TRIANGLE_LIST);
```

2. 现在我们需要在材质文件中创建这个材质。首先，我们将会定义着色器。Ogre 3D 需要有关着色器的五条信息。
   1. 着色器的名称。
   2. 用哪种语言写的。
   3. 在哪个文件中储存。
   4. 着色器的 main 函数是如何调用的。
   5. 我们需要编译着色器的哪些文件。

3. 所有这些信息应该出现在材质文件中。

```
fragment_program MyFragmentShader1 cg
```

```
{
   source Ogre3DBeginnersGuideShaders.cg
   entry_point MyFragmentShader1
   profiles ps_1_1    arbfp1
}
```

4. 顶点着色器需要同样的参数，但是我们也需要定义一个从 Ogre 3D 到着色器的参数。这包含了用于变换四边形到摄像机空间的矩阵。

```
vertex_program MyVertexShader1 cg
 {
   source Ogre3DBeginnerGuideShaders.cg
   entry_point MyVertexShader1
   profiles vs_1_1 arbvp1
   default_params
   {
      param_named_auto worldViewMatrix worldviewproj_matrix
   }
}
```

5. 材质本身仅使用顶点和片段着色器的名称来声明它们：

```
material MyMaterial13
{
      technique
      {
            pass
            {
                  vertex_program_ref MyVertexShader1
                {
                }
                fragment_program_ref MyFragmentShader1
                {
                }
```

```
        }
    }
}
```

6. 现在我们需要写着色器了。在你 Ogre 3D SDK 的 media\materials\programs 文件夹下创建一个名为 Ogre3DBeginnersGuideShaders.cg 的文件。

7 每个着色器看起来像一个函数。其中一点不同的是我们可以使用 out 关键字标识一个参数为传出参数来替代一个默认的传入参数。传出参数是在下一步中渲染线管使用。顶点着色器的传出参数被处理，然后传入一个像素着色器作为一个参数。像素着色器的传出参数是被用于创建最终的渲染结果。记住正确使用函数的名字；否则，Ogre 3D 就找不到它。让我们以片段着色器作为开始，因为它比较简单。

```
void MyFragmentShader1(out float4 color: COLOR)
```

8. 片段着色器将会返回蓝色作为每个像素渲染的颜色。

```
{
    color = float4(0，0，1，0);
}
```

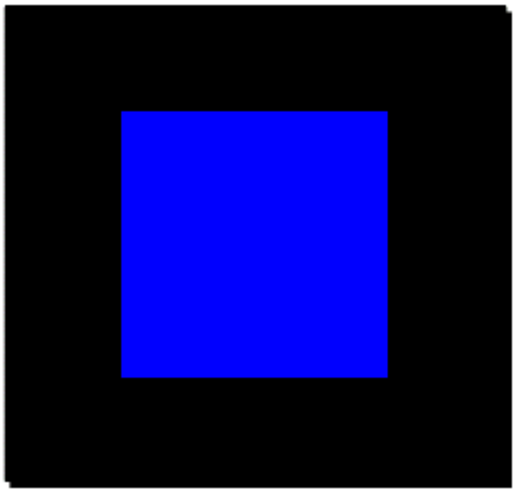
9. 这就是片段着色器；现在我们来到顶点着色器。顶点着色器有三个参数 —— 顶点的位置，顶点变换的位置作为我们的传出变量，并且作为我们用于变换矩阵 uniform 变量。

```
void MyVertexShader1(
        float4 position             : POSITION，
        out float4 oPosition        : POSITION，
        uniform float4x4 worldViewMatrix)
```

10 在着色器内部，我们使用了矩阵和传入的位置来计算传出位置：

```
{
    oPosition = mul(worldViewMatrix，  position);
}
```

11 编译运行程序。你应看到一四边形，这次它被渲染为蓝色。



刚刚发生了什么？

在这节发生了很多东西；我们将以第二步作为开始。这里，我们定义了将会使用的片段着色器。正如之前讨论过的，Ogre 3D 需要关于着色器的五条信息。我们用 *fragment_program* 关键字作为片段着色器的关键字，紧跟着就是我们给片段程序的函数名称，然后是一个空格，在这行最后，接着是写入所使用的着色器言语。对于程序而言，着色器是用汇编来写的，而且在早期，程序员必须用汇编写着色器代码，因为除了汇编没别的语言可用了。而且，对于一般的编程语言，高级语言的出现使写着色器代码变得容易了。目前，有三种不同的语言可以写入着色器：HLSL，GLSL 和 CG。着色语言 HLSL 被 DirectX 所使用，GLSL 是被 OpenGL 使用的语言。CG 语言是 Nvidia 和微软合作开发，并且它是我们将要使用的语言.这种语言在程序启动时被编译进各自的汇编代码。所以用 HLSL 写的着色器只能被 DirectX 支持，并且用 GLSL 编写的着色器只能被 OpenGL 所支持。但是 CG 可以编入 DirectX 和 OpenGL 的汇编代码。我们使用它的原因就是想跨平台。着色器的五条信息的两条是 Ogre 3D 所需要的，另外在大括号给出的三条，他的语法就好像一个属性文件——第一个是主键，然后是值。我们使用的主键后面跟着的是着色器文件的存贮位置。我们不需要给出完整的路径，仅需要文件名，因为 Ogre 3D 会扫描我们的目录并且根据需要的文件名来找到文件。

另一个我们使用的主键是 entry_point，紧随其后的是我们将要为着色器使用的函数名称。在代码文件中，我们创建了一个称为 MyFragmentShader1 的函数，并且我们把这个名称给交给 Ogre 3D 作为我们片段着色器的入口点。这意味着，每次我们需要片段着色器，这个函数就得到调用。这个函数只有一个传出参数 out float4 color : COLOR。这个前缀 out 表示这个参数是一个传出参数，意思是我们将会写入一个值，这个值将稍后被渲染线管所使用。这个参数的类型被称为 float4，表示有四个 float 值的数组。对于颜色，我们可以认为它是一四元组(r，g，b，a)，r 代表红色，g 代表绿色，b 代表蓝色和 a 代表 alpha 值：典型的描述颜色的元组。在参数类型之后，我们获得了一个: COLOR。在 CG 中，这被称作渲染线管上下文中被参数使用的语意描述。参数:COLOR 通知渲染线管此为一个颜色。在 out 关键字的组合下，这成为了一个片段着色器，从渲染线管可以推断出这是此片段着色器的颜色。

最后一条信息我们提供使用关键字 profiles 和它的值 ps_1_1 和 arbfp1。为便于理解，我们需要谈一些关于着色器历史。每一代的显卡，新一代的着色器被引入进来。它们开始时使用相当简单的没有 if 条件 C 语言风格的编程，但现在却用十分复杂而且强大的编程语言。并且现在存在有着色器的不同版本，而且每一个版本都有独特的函数集。Ogre 3D 需要知道它使用的是哪个版本。ps_1_1 表示像素着色器的版本是 1.1，arbfp1 表示片段着色器的版本是 1。我们需要两个档案资料，因为 ps_1_1 是 DirectX 特有的函数集，而 arbfp1 是 OpenGL 的子函数集。所有的子集可以在 http://www.ogre3d.org/docs/manual/manual_18.html 找到。这都需要在材质文件中定义片段着色器。在第三步，我们定义了我们的顶点着色器。这部分和定义的片段着色器代码非常相似；最主要的不同就是 default_params 那段代码。这段定义了在执行期给着色器的参数。param_named_auto 定义了一个被 Ogre 3D 自动传给着色器的参数。在这个关键字之后，我们需要给参数一个名称，而且在此之后，我们想要它拥有关键字的值。我们命名参数为 worldViewMatrix；别的名字也会同样起作用，并且我们想要它拥有关键字 worldviewproj_matrix. 这个关键字告诉 Ogre 3D 我们想要我们的参数拥有 WorldViewProjection 矩阵的值。这个矩阵的作用是把局部空间的顶点变为摄像机空间的顶点。所有关键字值得列表可以在 http://www.ogre3d.org/docs/manual/manual_23.html#SEC128 找到。我们如何使用这些值你将会稍后看到。

在第四步中使用了之前写过的代码。一如往常，我们用一种技术和一个 pass 定义了我们的材质，我们没有定义一个纹理单元，但是使用了关键字 vertex_program_ref。在这个关键字之后，我们需要给它一个定义好的顶点程序的名称，在我们的案例中，这个名称是 MyVertexShader1。如果需要，我们可以把更多的参数放进定义中，但是我们并不需要，所以我们仅用大括号来开始和关闭代码段。这个规则对于 fragment_program_ref 也是适用的。

写出一个着色器

现在我们定义好了材质文件的所有必要的东西，让我们自己写一个着色器程序。在第六步定义了我们之前讨论过的函数的参数表，所以我们不再深入讲解。第七步定义了函数体；对于片段着色器，函数体很简单。我们创建了一个新的 float4 元组(0，0，1，0)，表示了蓝色并把颜色赋值到参数中。这个效果就是所有用此材质渲染的东西都是蓝色的。所有片段着色器要讲的就这些了，让我们把注意力放到顶点着色器上。在第八步定义了函数的头部。顶点着色器有 3 个参数——两个使用 CG 语法标记位置，另一参数是使用 float4 作为 worldViewMatrix 值的一个 4*4 的矩阵。在参数类型定义之前，有 uniform 的关键字。

每次调用我们的顶点着色器，它获得一个新顶点作为输入的位置参数，计算新顶点的位置，保存它到 oPosition 参数。这表示每次调用函数，参数都会改变。而 worldViewMatrix 却不改变。关键字 uniform 表示参数在每次绘图调用是恒定的。当我们渲染四边形时，worldViewMatrix 不改变，而其他参数会因每次顶点着色器所处理顶点的不同而不同。在第九步，创建了顶点着色器的函数体。在函数体中，我们乘以从世界矩阵获得的顶点来得到摄像机空间中的顶点。这个变换的顶点保存在由渲染线管处理的传出参数中。在我们做更多有关着色器实验后，我们会更近一步的了解渲染线管。

7.12 纹理着色

我们用蓝色作为四边形的输出，但我们更愿意用之前的纹理。

实践时刻——在着色器中使用纹理

1. 创建一个称为 MyMaterial14 的新材质。同样创建两个新的称为 MyFragmentShader2 和 MyVertexShader2 的着色器。记住复制在材质文件中的定义过的片段和定点着色程序。添加一个岩石纹理单元到材质文件：

```
texture_unit
{
   texture terr_rock6.jpg
}
```

2. 我们需要添加两个新的参数到我们的片段着色器中。第一个是一关于纹理坐标的 float 类型的二元组。因此，我们同样使用语意来标志此参数为我们使用的第一个纹理坐标，另一个新的参数的类型是 sampler2D，这是纹理的另一个名称。因为纹理不改变的基础上，每一个片段，我们标志为 uniform。这个关键字表示参数值来自于外部的 CG 程序，并且它是由渲染的环境设置的，在当前情况下，是由 Ogre 3D 设置的。

```
void MyFragmentShader2(float2 uv          : TEXCOORD0，
                       out float4 color : COLOR，
                       uniform sampler2D texture)
```

3. 在片段着色器中，把颜色赋值用下面一行代码替代。

```
color = tex2D(texture，  uv);
```

4. 在顶点着色器中同样需要一些新的参数——一个 float2 来接受传入的纹理坐标，一个 float2 作为传出纹理坐标。两个纹理坐标都是 TEXCOORD0，因为传入的坐标和另一个传出的坐标都为 TEXCOORD0：

```
void MyVertexShader2(
        float4 position          : POSITION，
        out float4 oPosition     : POSITION，
        float2 uv          : TEXCOORD0，
```

```
        out float2 oUv          : TEXCOORD0，
        uniform float4x4 worldViewMatrix)
```

5. 在函数体中，我们计算顶点的传出坐标：

```
oPosition = mul(worldViewMatrix，  position);
```

6. 对于纹理的坐标，我们赋值传入的值到传出：

```
oUv = uv;
```

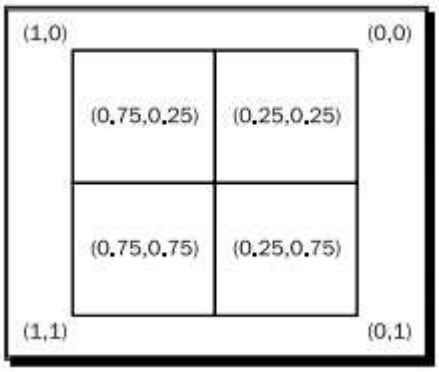
7. 记住改变在程序中材质的代码，然后编译运行它。你将会看到四边形有一个岩石的纹理。

刚刚发生了什么？

在第一步中仅给一纹理单元添加了岩石纹理，没什么特别。在第二步中添加了一个 float2 来保存纹理坐标信息；同样我们第一次使用了 sampler2D。sampler2D 是一个二维纹理查找函数的名称，而且因为它没有改变每个片段，而且来自于外部的 CG 程序，我们声明它为 uniform。在第三步中，使用了 tex2D 函数，此函数使用接收一个 sampler2D 和 float2 参数，并返回一个类型为 float4 的颜色。这个函数使用 float2 作为位置来检索 sampler2D 对象的颜色并返回这个颜色。基本上，它仅是用给定的纹理坐标在纹理中查找。在第四步中，添加两个纹理坐标到顶点着色器——一个是传入参数，一个是传出参数。在第五步赋值传入参数到传出参数。这些奇妙的处理发生在渲染线管中。

在渲染线管中发生了什么？

我们的顶点着色器获取每个顶点并变换它到摄像机空间。在所有的顶点通过这种变换之后，渲染线管了解到哪个顶点来自三角形，然后光栅化它们。在这个过程中，三角形被分为很片段。每个片段都成为屏幕上显示像素区域的缓冲区域。如果它们没有被别的片段所覆盖的话，它们将会显示出来。反之，就会看不到。在此过程中，渲染线管插入顶点数据，就如纹理坐标遍历每个片段一样。在此过程之后，每个片段都有了自己纹理坐标，而且我们使用这种方法在查找纹理中的颜色值。每个片段有自己的纹理坐标。这也显示出我们如何把纹理坐标关联到点的。在现实世界中，这种依附于渲染线管的方式是可变的，虽说这种模型有助于我们理解，但也不是百分百准确的。



当赋值给每个顶点一个颜色时，我们使用了相同的修改。现在让我们更深入的了解这种效果吧。

动手试试 ——把颜色和纹理坐标联合起来

分别创建一个新的 MyVertexShader3 顶点着色器和称为 MyFragmentShader3 片段着色器。片段着色器应该渲染所有的东西为绿色，顶点着色器应计算出在摄像机空间顶点的位置，并简单的把纹理坐标传递给片段着色器。片段着色器目前还对它们做不了任何事，但是我们稍后会用到它。

7.13 改变颜色

为了使修改的效果看起来更明显，让我们以颜色代替纹理。

实践时刻 —— 使用颜色来观察改变

研究一下颜色的修改是如何起作用的，我们现在需要修改一下代码.

1. 同样的，复制关于材质文件并确保校正了所有的名字。

2. 我们唯一需要做的就是在材质文件中删除我们不需要的纹理。我们可以直接删除它。

3. 在程序的代码中，我们需要以 *color()* 函数来代替 *textureCoord()* 函数

```
manual->position(5.0， 0.0， 0.0);
manual->color(0，0，1);
manual->position(-5.0， 10.0， 0.0);
manual->color(0，1，0);
manual->position(-5.0， 0.0， 0.0);
manual->color(0，1，0);
manual->position(5.0， 10.0， 0.0);
manual->color(0，0，1);
```

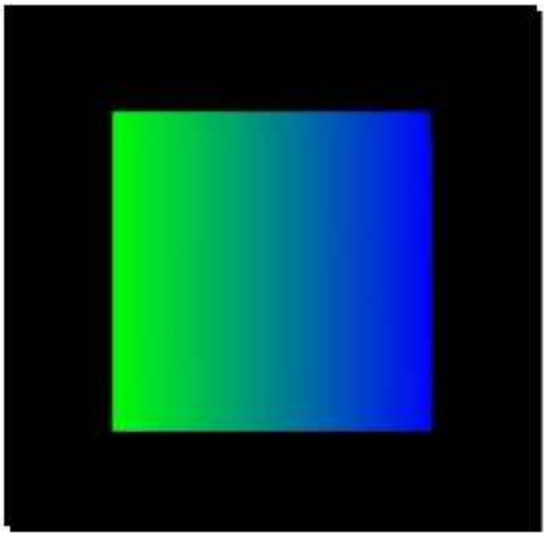
4. 在顶点着色器中也需要一些修改。用颜色参数替换两个纹理坐标的参数并同样修改赋值的那一行：

```
void MyVertexShader4(
        float4 position          : POSITION，
        out float4 oPosition     : POSITION，
        float4 color     :COLOR，
        out float4 ocolor     :COLOR，
        uniform float4x4 worldViewMatrix)
{
    oPosition = mul(worldViewMatrix， position);
    ocolor = color;
}
```

5. 片段着色器现在有两个颜色参数 —— 一个是传入参数和另一个传出参数：

```
void MyFragmentShader4(  float4 color : COLOR，out float4 oColor : COLOR)
{
    oColor = color;
}
```

6. 编译并运行程序。你可以看到一个右边是蓝色，左边是绿色，中间为渐变的四边形。



刚刚发生了什么？

在第三步，我们看到另一个 manual 对象的成员函数，使用三个浮点型的代表红色，绿色和蓝色来添加颜色到一个顶点。第四步中替换纹理坐标为一个颜色参数 —— 这次我们想要的是颜色而非纹理。第五步也是同理的。这个例子并不是特别困难和令人感到惊喜，但是它显示出修改是如何起作用的。这会使我们对顶点和片段着色器有个更好的理解。

7.14 用模型替代四边形

四边形作为实验的例子显得略微无聊，那么让我们用 Sinbad 的模型来替换他。

实践时刻 —— 用模型来代替四边形

使用之前的代码我们将会使用 Sinbad 来代替四边形。

1. 删除关于创建四边形的所有代码；仅仅留着创建场景结点的代码。

2. 创建一个 *Sinbad.mesh* 的实例，并管理他到场景结点，并使用 *MaterialManager*（材质管理器）来设置实体的材质到 *MyMaterial14*：

```
void createScene()
{
   Ogre::SceneNode* node = mSceneMgr->
```

```
                                    getRootSceneNode()->createChildSceneNode("Node1");
    Ogre::Entity* ent = mSceneMgr->createEntity("Entity1"， "Sinbad.mesh");
    ent->setMaterialName("MyMaterial14");
    node->attachObject(ent);
}
```

3. 编译并运行程序；因为 *MyMaterial14* 使用了颜色纹理，Sinbad 将会看起来是用岩石铸成的。

刚刚发生了什么？

这里刚刚添加的代码对于现在你来说应该已经轻车熟路了。我们创建了一个模型的实例，并关联他到场景结点上，然后改变材质为 MyMaterial14。

7.15 使模型在 X 轴上振动

目前为止，我们仅仅使用了片段着色器。现在是时候让我们使用顶点着色器了。

实践时刻 —— 添加一个振动

添加一个振动到我们的模型是十分简单的并且仅需要改变一些我们的代码。

1. 这次，我们仅需要一个新的顶点着色器因为我们将会使用已有的片段着色器。创建一个名为 *MyVertexShader5* 新的顶点着色器并在新的材质 *MyMaterial17* 使用它，仅使用 MyFragmentShader2，是因为这个着色器在我们的模型上添加了纹理。

```
material MyMaterial17
{
   technique
   {
         pass
         {
      vertex_program_ref MyVertexShader5
               {
               }
            fragment_program_ref MyFragmentShader2
               {
               }

      texture_unit
               {
                     texture terr_rock6.jpg
               }
         }
   }
```

2. 这个新的顶点着色器和我们之前看到过的着色器近乎相同；仅仅在 default_params 代码段中添加了一个新的参数，称为 pulseTime。它能从时间关键字中获取值：

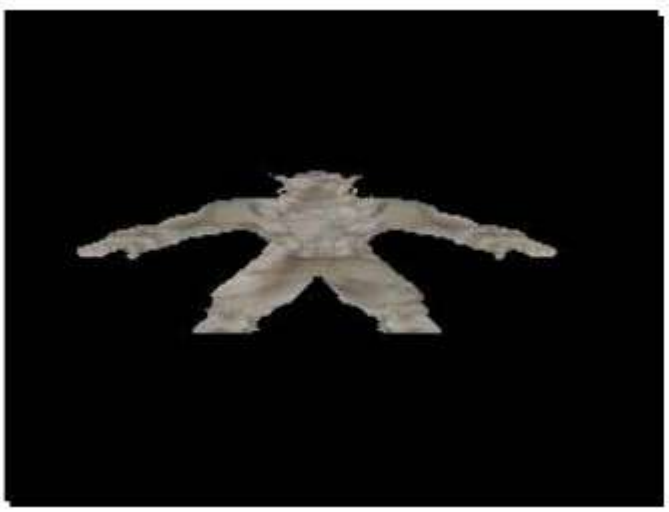
```
vertex_program MyVertexShader5 cg
 {
   source Ogre3DBeginnerGuideShaders.cg
   entry_point MyVertexShader5
   profiles vs_1_1 arbvp1
   default_params
   {
```

```
        param_named_auto worldViewMatrix worldviewproj_matrix
        param_named_auto pulseTime time
    }
}
```

3. 我们不需要修改程序本身的代码（译者注：实际需要修改一下 setMaterialName 的参数）。我们所需要做的最后一件事就是创建一个新的顶点着色器。MyVertexShader5 是基于 MyVertexShader3 的。仅添加新的一行来使 oPosition 变量的 x 值乘以(2+sin(pulseTime))：

```
void MyVertexShader5( uniform float pulseTime，
            float4 position             : POSITION，
            out float4 oPosition        : POSITION，
            float2 uv             : TEXCOORD0，
            out float2 oUv              : TEXCOORD0，
            uniform float4x4 worldViewMatrix)
{
    oPosition = mul(worldViewMatrix，  position);
    oPosition.x *= (2+sin(pulseTime));
    oUv = uv;
}
```

4. 编译运行程序。你将会看到 Sinbad 在 x 轴方向振动，振幅在他的普通宽度和三倍宽度之间。



刚刚发生了什么？

我们使得模型在 X 轴上有了个振动。我们需要给顶点着色器包含目前时间的第二个参数。我们使用了时间的正弦值以得到 1 到 3 之间的值(译注:1~3 是由 sin 范围的[-1，1]+2 = [1，3]得来)，使得我们模型顶点的 x 部分乘以变化的正弦值。创建了一个振动的效果。使用这种技术，我们可以传递任何信息到一个着色器中，以此来修改它的行为。这是在很多游戏中已使用特效的一个原型。

7.16 小结

我们这章学习了很多有关材质和 Ogre 3D 的知识。

具体来说，我们的内容涵盖了：

* 如何创建新的材质。
* 如何使用材质应用到实体上。
* 如何创建着色器和如何在材质文件中引用它们。
* 渲染线管是如何工作的和如何使用顶点着色器来修改几何模型。

在下一章，我们将会创建 *post-processing* 效果以提升我们场景的视觉质量和创建崭新的视觉风格。

第八章 合成器框架

在这一章，我们将会添加后期处理的特效到场景之中，这可以提升场景的视觉效果并且使他们看起来更有趣。这一章将会展示给你如何创建合成器和如何结合它们创建新的特效。

这一章，我们将会:

创建合成器脚本并应用他们到我们的场景
使用视口来创建分屏
使用用户输入并使用合成器操作着色器参数，

那么，让我们开始吧….

8.1 准备一个场景

我们将会使用合成器特效。但是，在使用他们之前，我们将会准备一个场景，这样就可以展示出我们创建的不同特效了。

实践时刻 —— 准备场景

我们将会使用上一章的最后一个例子：

1. 删除改变模型材质的一行代码。我们想要它使用其本身的材质：

```
ent->setMaterial(Ogre::MaterialManager::getSingleton().getByName("MyMaterial18"));
```

2. 类中的代码现在应该看起来如下面一样：

```
class Example69 : public ExampleApplication
{
private:

public:
   void createScene()
   {
    Ogre::SceneNode* node = mSceneMgr->getRootSceneNode()
```

```
                    ->createChildSceneNode("Node1"，Ogre::Vector3(0，0，450));
    Ogre::Entity* ent = mSceneMgr->createEntity("Entity1"，"Sinbad.mesh");
    node->attachObject(ent);
  }
};
```

3. 编译运行程序。你将会看到 Sinbad 本身材质的渲染实例。

刚刚发生了什么？

我们创建一个接下来要使用合成器的简单场景。

添加第一个合成器

在解释什么是合成器之前，让我们先使用一个然后再讨论技术细节。

1. 我们需要一个新的材质，但是它目前不起作用。添加这个新的材质到之前使用过的材质文件中，并命名为 *Ogre3DBeginnersGuide/Comp1*：

```
material Ogre3DBeginnersGuide/Comp1
{
  technique
    {
        pass
        {
```

```
    texture_unit
            {
            }
        }
    }
}
```

2. 然后，创建一个新的文件来存储我们的合成器脚本。和材质文件是同一个目录，创建一个文件命名为 *Ogre3DBeginnersGuide.compositor* ：

3. 在这个文件里，使用和材质同样的方案来定义我们的合成器:

```
compositor Compositor1
{
    technique
{
```

4. 接着，在修改之前定义一个目标以指示我们的渲染的场景

```
texture scene target_width target_height PF_R8G8B8
```

5. 定义我们目标的内容。在当前的情况下，即为我们之前渲染的场景：

```
target scene
        {
            input previous
        }
```

6. 合成器脚本的最后一步为定义输出：

```
target_output
        {
```

7. 合成器不需要任何输入并把其结果渲染到一个覆盖整个窗口的四边形上。这个四边形使用材质 *Ogre3DBeginnersGuide/Comp1* 并需要我们的场景目标作为一个纹理输入：

```
input none
pass render_quad
{
material Ogre3DBeginnersGuide/Comp1
input 0 scene
}
```

8. 这样这个合成器就结束了。封闭所有的花括号：

```
        }
    }
}
```

9. 现在我们已经完成了一个合成器脚本，让我们添加它到场景中。为到达目地，我们使用 *CompositorManager*(合成器管理器)和我们摄像机的视口。添加代码到 *createScene()*函数中：

```
Ogre::CompositorManager::getSingleton().addCompositor(
                                        mCamera->getViewport(),  "Compositor1");
Ogre::CompositorManager::getSingleton().setCompositorEnabled(
                                    mCamera->getViewport(),  "Compositor1",  true);
```

10. 编译运行程序，你将会看到和之前一样的场景。

刚刚发生了什么？

我们使用一个包含合成器脚本的合成器文件添加了我们的第一个合成器。第一步仅创建了一个空的材质，这个材质得到渲染的信息，但并不添加任何特效。第二步创建了一个新的文件来存储我们的合成器脚本；就像材质文件一样，不过这个是合成器。第三步我们就比较熟悉了：我们命名我们第一个合成器为 Compositor1，并如材质文件中定义的样式，使用一种 technique，合成器对于不同的显卡有不同的 techniques。从第四步开始就比较有趣了：这里我们创建了一个名为 scene 的新纹理，它和渲染的目标纹理有同样的大小，并且这个纹理对每个颜色分量使用 8 个位。这种格式化像素的方式由 PF_R8G8B8 来定义。

合成器是如何起作用的

但是为什么我们需要首先创建一个新的纹理呢？为理解这个，我们需要理解合成器的工作原理。合成器是在场景渲染之后来修改场景的外观。这个过程就好像电影中的后期处理，

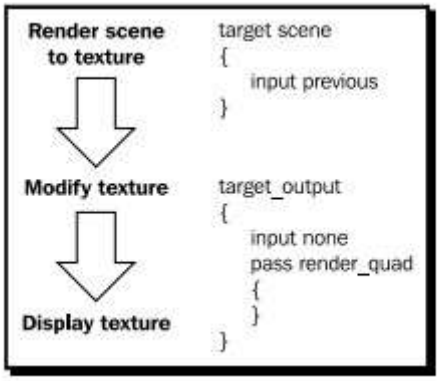
所谓后期处理就是电影杀青后添加电脑特效。为了实现这个任务，合成器需要把渲染后的场景作为一纹理，这样它就可以被修改了。我们在第四步创建了这个纹理，并且在第五步，我们告诉 Ogre 3D 以场景之前的渲染来填充这个纹理。这是由 input previous 来完成的。现在我们有一个包含我们渲染后场景的纹理，我们下一个场景管理器需要做的就是创建一个新的图像，这个图像最终被在显示设备上显示。这一过程是在第六步使用关键字 output 和一个目标代码段来完成的。

在第七步定义这个输出。我们不需要任何输入，因为场景纹理已有场景图像。为渲染修改的纹理，我们需要一个四边形来覆盖整个显示设备，我们可以在这个设备渲染已经修改了的场景。这是用 pass 关键字后的 *render_quad* 标识符来实现的。这里也有很多别的可以放在 pass 关键字后标识符。他们可以在文档地址 (http://www.ogre3d.org/docs/manual/manual_32.html#SEC163) 处找到。

在 pass 的代码段中，我们定义了用于渲染传递的几种不同的属性。第一个是我们四边形将要使用的材质；这里我们仅使用我们提前定义好的材质，这个材质无修改渲染四边形的纹理。下一个属性定义了如纹理类似的附加输入；这里我们想要输入的第一纹理作为场景纹理。实际上，这表示我们想要我们的场景渲染到一个纹理并应用至一个覆盖整个屏幕的四边形。在这里，我们将看不到加不加合成器对渲染的场景有什么不同。当我们添加一些代码到材质中就会有很大的改观了，添加的代码将会修改传入的纹理并添加附加的特效】

第九步添加合成器到我们的视口并激活它；我们添加合成器到一个视口而非我们的场景是因为合成器通过摄像机修改了场景的景象，而摄像机的视图是与视口相关的。所以如果我们想要修改整个场景的景象，我们应添加合成器到对象，对象定义了场景本身的景象。

下面的简化图示显示出了合成器和一删简版的合成器脚本的工作流程，图示显示了代码每一步所代表的流程。



8.2 修改纹理

我们渲染我们的场景到一个纹理，但显示的效果却没有什么变化。这是相当无趣的。那么让我们把它变得有趣一点吧。

实践时刻 —— 修改纹理

1. 我们将会使用一个片段着色器并修改纹理，所以修改材质使用片段着色器。同样的复制材质和合成器。这个合成器的名字应为 *Compositor2*，材质名 *Ogre3DBeginnersGuide/Comp2*：

```
fragment_program_ref MyFragmentShader5
                    {
                    }
```

2. 在使用引用之前不要忘记在纹理文件中定义片段着色器

```
fragment_program MyFragmentShader5 cg
{
   source Ogre3DBeginnersGuideShaders.cg
   entry_point MyFragmentShader5
   profiles ps_1_1    arbfp1
}
```

3. 同样在着色器文件中创建一个新的片段着色器。就输入而言这个着色器有纹理坐标和纹理样例。当片段的颜色被计算的时，颜色将会返回：

```
void MyFragmentShader5(float2 uv : TEXCOORD0，
                out float4 color : COLOR，
                uniform sampler2D texture)
{
```

4. 在纹理坐标系的位置获取纹理的颜色：

```
float4 temp_color = tex2D(texture，   uv);
```

5. 转变颜色为灰度模式：

```
float greyvalue = temp_color.r * 0.3 + temp_color.g * 0.59 + temp_color.b * 0.11;
```

6. 并使用这个值作为三个输出颜色分量的值：

```
color = float4(greyvalue， greyvalue， greyvalue， 0);
```

7. 编译运行程序。你将会看到同样的模型，但是这次是在灰度模式以下的：

刚刚发生了什么？

我们添加一个片段着色器到我们的合成器中。在渲染过我们的场景之后，整个窗口四边形通过场景纹理使用我们的片段着色器得到渲染。片段着色器通过使用他的纹理坐标来查询坐标位置的颜色。然后程序用红色分量的 0.3，绿色分量的 0.59，蓝色分量的 0.11 转换 RGB 颜色值为灰度模式下的颜色。这些值代表每种颜色分量对于人眼接受亮度的影响。 使用这些值添加到颜色创建了和灰度显示比较接近的效果。在这个合成器特效之后，我们将会创建一种并非转变图像为黑白色的效果，但效果要比转化颜色更胜一筹。】

8.3 转换图片

现在是时候用另一个合成器来创建场景颜色转换的另一个版本了。

实践时刻 —— 转换图片

使用黑白模式合成器的代码，我们现在将要转换图片。

1. 复制着色器，材质和合成器因为稍后我们将会需要黑白模式着色器和当前的这个着色器。然后改变这个改变这个复制的片段着色器。这个新的片段着色器应命名为 *MyFragmentShader6*，材质命名为 *Ogre3DBeginnersGuide/Comp3*，作为合成器 *Compositor3*。

2. 这时，获取纹理片段位置的颜色值，然用 1.0 减每个颜色分量来获取转换的颜色值：

```
color = float4( 1.0 - temp_color.r，1.0 - temp_color.g，  1.0 - temp_color.b，0);
```

3. 编译运行程序。这次，背景将会是白色的，Sinbad 将会如下图显示的那样，其身上是比较奇怪的颜色：

刚刚发生了什么？

我们改变了片段着色器以替换他们之前转换使用的黑白模式。程序的其他部分没有什么改变。转换 RGB 颜色值是比较简单的：仅用他们颜色分量的最大值减去当期颜色分量，在我们当前情况下是 1.0。最终的颜色是原始颜色的反转。

8.4 结合合成器

这么快变的略显无聊了，那么让我们结合两种着色器特效吧。

实践时刻 —— 结合两种合成器

为结合两个合成器，我们需要创建一个新的合成器：

1. 为创建一个新的合成器，我们需要两个纹理——一个来存储场景并且一个来存储一些临时的结果

```
compositor Compositor4
{
    technique
     {
          texture scene target_width target_height PF_R8G8B8
          texture temp target_width target_height PF_R8G8B8
```

2. 如之前一样填充场景纹理，然后使用场景纹理和我们的黑白模式的纹理来填充 temp 纹理。

```
target scene
        {
             input previous
        }
        target temp
    {
          pass render_quad
          {
            material Ogre3DBeginnersGuide/Comp2
            input 0 scene
           }
        }
```

3. 然后使用临时材质和我们的反转材质来创建输出纹理：

```
target_output
          {
               input none
               pass render_quad
               {
```

```
                material Ogre3DBeginnersGuide/Comp3
                input 0 temp
            }
        }
    }
}
```

4. 编译和运行程序；你将会看到看到一个先颜色转换至黑白色，然后颜色反转的场景。

刚刚发生了什么？

我们创建了第二个辅助纹理；这个纹理作为一个渲染目标设为黑白模式，然后这个纹理就会有我们场景的黑白图像作为反转材质的输入，处理之前图像才得以显示。

8.5 减少纹理总数

在之前的部分，我们使用两个纹理—— 一个是原始场景，另一个是在两步改变中的第一步完成之后来存储中间结果。现在让我们尝试仅使用一种纹理。

实践时刻 —— 减少纹理总数

通过使用之前的代码，我们将会减少我们合成器的纹理总数。

1. 我们需要一个新的合成器，这次只有一个纹理：

```
compositor Compositor5
{
    technique
     {
          texture scene target_width target_height PF_R8G8B8
```

2. 然后用渲染的场景填充纹理：

```
target scene
{
    input previous
}
```

3. 使用纹理作为输入纹理同时也为输出纹理：

```
target scene
{
   pass render_quad
   {
     material Ogre3DBeginnersGuide/Comp2
     input 0 scene
   }
}
```

4. 同样的，使用使用纹理作为输入作为最终渲染：

```
target_output
{
   input none
   pass render_quad
   {
     material Ogre3DBeginnersGuide/Comp3
     input 0 scene
   }
```

```
}
```

5. 添加缺少的括号：

```
    }
}
```

6. 编译运行程序。效果是一样的，但是我们这次仅使用使用了一个纹理。

刚刚发生了什么？

我们仅使用一个纹理改变了我们的合成器，并且发现在合成器中可以不止一次的使用纹理。我们发现可以同时使用纹理作为输入和输出。

动手试试 —— 交换绿色和蓝色通道

创建一个合成器来交换场景的绿色和蓝色通道。结果将会看下来如下图：

8.6 一些更复杂的东西

目前位置，我们只看到了比较简单的合成器。那么让我们写一个更复杂的吧。

实践时刻 —— 复杂的合成器

我们需要一个新的合成器，材质和片段着色器：

1. 合成器脚本本身并没有什么特别的。我们需要场景的一个纹理，然后直接使用这个纹理作为输入来关联输出，输出使用了一个材质：

```
compositor Compositor7
{
    technique
     {
          texture scene target_width target_height PF_R8G8B8
          target scene
          {
               input previous
          }

          target_output
          {
               input none
               pass render_quad
               {
                    material Ogre3DBeginnersGuide/Comp5
                    input 0 scene
               }
          }
    }
}
```

2. 材质本身并没有什么特别；并如往常一样仅添加一个片段着色器：

```
material Ogre3DBeginnersGuide/Comp5
{
   technique
      {
```

```
            pass
            {
        fragment_program_ref MyFragmentShader8
                    {
                    }

        texture_unit
                    {
                    }
            }
        }
}
```

3. 不要忘记在使用材质之前添加片段着色器的定义：

```
fragment_program MyFragmentShader8 cg
{
    source Ogre3DBeginnersGuideShaders.cg
    entry_point MyFragmentShader8
    profiles ps_1_1    arbfp1
}
```

4. 现在到比较有趣的部分了。在片段着色器中，自从最后一次算起函数头没有什么改变；只有函数体改变了。首先，我们需要两个变量，称为，num 和 stepsize。变量 stepsize 是 1.0 除以 num 的结果：

```
float num= 50;
float stepsize = 1.0/ num;
```

5. 然后使用两个变量和纹理坐标来计算新的纹理坐标：

```
float2 fragment = float2(stepsize * floor(uv.x * num)，stepsize *    floor(uv.y * num));
```

6. 使用新的纹理坐标来从纹理中检索颜色：

```
color = tex2D(texture， fragment);
```

7. 改变程序仅使用这个着色器，而不结合别的合成器。然后编译运行程序。你应看到几种不同颜色的像素替代了 Sinbad 正常的实例。

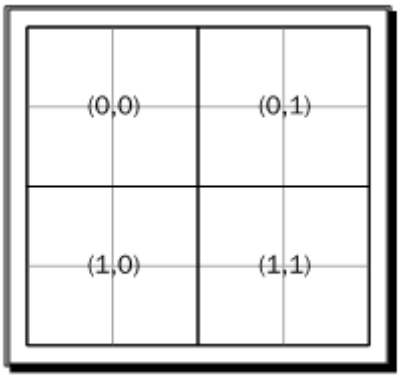
(译注* 不同的显卡和渲染方式因为计算浮点的方式不同，很可能显示的效果不同，如果没有看到上图的效果，可以尝试用 OpenGL 的方式渲染程序。)

刚刚发生了什么?

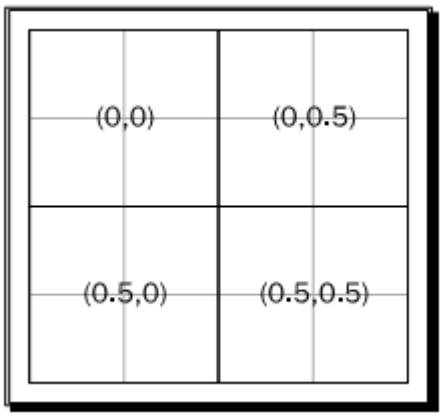
我们创建了另一个可以改变场景外观的合成器。这个效果几乎不可辨认模型。在第一步和第二步我们已经很熟悉了，应该理解起来没有什么困难。在第四步(译注：原文错为第三步，原文类似错误后面改正后不再提醒) ，设置了我们稍后需要使用的值。一是我们将要的像素尺寸的大小，所以我们设值为 50.这表示每个轴，x 和 y，将会有 50 像素。第二个值是 *stepsize*，这个值是用 1 除以像素的数量。我们需要这个值来计算纹理的坐标。

在第五步，我们使用旧的纹理坐标和我们两个之前定义的两个值来计算出新的纹理坐标。那么我们如何使用片段着色器来减少像素的数目呢？此种解决方式的显示结果将会为 100*100 分辨率。如果用这种分辨率来渲染场景，场景将会看起来很正常而不会看到在之前例子中的单个像素。为使用更少的显示像素以获取这种效果，我们需要几个相邻的像素为同样的颜色。我们在五步中使用了一个简单的算法实现了这种效果。第一步是用原始坐标乘以我们最后想要的像素数目，然后下一步把这个浮点型值四舍五入为低位整型。这将会给出和这个像素相同的最终数目。

比如说我们场景为 4*4 分辨率，我们想要最终的图像只有 2*2。如果我们有一原始的纹理坐标(0.1，0.1)，我们用最终的矩阵分辨率乘以他们，然后得到(0.2，0.2)。然后我们四舍五入这些值为(0，0)的低位整型。这告诉我们此像素的最终像素为(0，0)。如果有一像素的坐标为(0.75，0.75)，结果将会为(1.5，1.5)并且四舍五入为(1，1)。通过这个简单的操作，我们可以计算出原始像素的最终像素。



在我们知道元素像素的最终像素后，我们需要计算纹理的坐标以从原始场景纹理上检索颜色值。为实现这个，我们需要我们第二个称为 stepsize 的值。我们例子的 stepsize 的值为 0.5，因为我们用 1 除以我们第二步预定好的像素值。然后我们用 stepsize 乘以不同的纹理坐标值，以获得最终的纹理坐标。



我们使用这些值从场景纹理中检索颜色值并使用这些值作为新的颜色值。我们这样做是因为四个像素有同样的颜色值，这样看起来就好像一个大的像素一般。这使得一个场景比他实际的大小有更少的像素。当然，这种技术并不完美。一个更好的方法是计算原始像素的总体平均值以计算最终像素。

8.7 改变像素数量

我们现在有一个合成器可以使我们的场景看起来比实际的像素要少，但是像素的数量在片段着色器中是硬编码。当我们想要以不同的像素来使用合成器时，我们需要创建一个新的合成器，材质和片段着色器，这不是有效率的做法。第一步我们要完成的是在材质中定义像素数量而非在片段着色器中。这样，我们至少可以重用不同的数量像素着色以完成我们最终想要的图像。

实践时刻 —— 放置像素数量至材质

现在我们将会从材质来控制像素的数量而不是从着色器本身。

1. 创建一个新的有着所有旧参数的片段着色器，但多了一为 uniform 的新参数，称为 numpixels 的浮点型形参：

```
void MyFragmentShader9(float2 uv : TEXCOORD0，
        out float4 color : COLOR，
        uniform sampler2D texture，
        uniform float numpixels)
{
```

2. 然后，在片段着色器内部，使用新的参数来设置 num 变量:

```
float num = numpixels;
```

3. 着色器的其他部分和之前的片段着色器相同

```
float stepsize = 1.0/num;

float2 fragment = float2(stepsize * floor(uv.x * num)，stepsize *  floor(uv.y * num));
  color = tex2D(texture，  fragment);
}
```

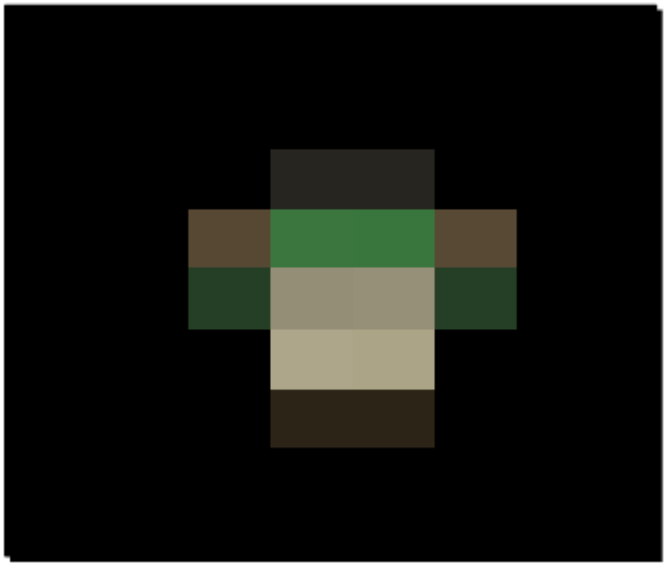
4. 此处的材质是一几乎和原来相同的副本。只有片段着色器的声明需要改变一下；或者更准确的说，需要添加一些新的东西到 default_params 代码段。在这段中，定义一个 numpixels 参数，这个参数为 float 型并且值为 500：

```
fragment_program MyFragmentShader9 cg
{
   source Ogre3DBeginnerGuideShaders.cg
   entry_point MyFragmentShader9
   profiles ps_1_1    arbfp1

   default_params
   {
    param_named numpixels float 500
   }
}
```

5. 现在创建一个使用新材质的新合成器，并改变程序代码使用新的合成器。稍后，编译运行程序。你将会看到一个被渲染 Sinbad 的实例，这个实例比原来的模型的分辨率要小。

6. 现在改变 numpixels 的值为 25 并再次运行程序。不需要重新编译，因为我们改变的是一个脚本而不是代码文件。



刚刚发生了什么？

我们把像素数目从片段着色器中放到了想要的材质中，这样就可以允许我们无需改变和复写片段着色器来改变像素数目。

为简单的实现这个目地；我们仅需要添加一个新的 uniform 变量到片段着色器中并添加一个 default_params 代码到材质中，原本我们在材质的代码段中是需要定义变量名，类型和 param_named 关键字的值的。对于 Ogre 3D，为了能够映射材质中的参数到片段着色器中，保持型别和名字相同是十分必要的。

动手试试 —— 尝试不同的像素数量

以 numpixels 的值为 50，100 和 1000 来运行程序。

8.8 在代码中设置变量

我们把 numpixels 变量从片段着色器代码移动到材质脚本中。现在，让我们尝试从程序代码中设置值。 】

实践时刻 —— 从程序中设置变量

我们可以使用上个例子的合成器，材质和片段着色器。只有需要程序本身修改。

1. 我们不能直接作用于渲染的四边形材质，因为我们程序不知道那个四边形。监听是

唯一可以影响合成器渲染过程的途径。Ogre 3D 提供了一个合成器监听的接口。我们可以使用这个来创建一个新的监听：

```
class CompositorListener1 : public Ogre::CompositorInstance::Listener
{
public:
```

2. 覆写这个在当材质建立时调用的方法：

```
void notifyMaterialSetup (uint32 pass_id， MaterialPtr &mat)
{
```

使用获取到的材质指针来改变 *numpixels* 参数为 125 ：

```
   mat->getBestTechnique()->getPass(pass_id)->
                getFragmentProgramParameters()->setNamedConstant("numpixels"， 125.0f);
}
```

4. 在添加完激活合成器的代码后，在 createScene()函数中添加下面的代码以获得合成器的实例

```
Ogre::CompositorInstance* comp =  Ogre::CompositorManager::getSingleton()
                                      getCompositorChain(mCamera->getViewport())->
                                      getCompositor("Compositor8");
```

5. 我们需要一个 private 变量来稍后存储监听实例：

```
private:
   CompositorListener1* compListener;
```

6. 当程序创建的时候，我们需要把这个指针设为 NULL ：

```
Example78()
{
   compListener = NULL;
}
```

7. 我们创建的对象，我们也需要摧毁它。添加一个摧毁监听实例的析构函数到程序中：

```
~Example78()
  {
   delete compListener;
  }
```

8. 现在创建一个监听并添加它到合成器实例：

```
compListener = new CompositorListener1();
comp->addListener(compListener);
```

9. 编译运行程序。 你将会看到有一个有些许像素的场景，如下图显示的一般：

刚刚发生了什么？

我们改变了我们的程序，这样就可以从程序代码中修改像素数目而不是从材质脚本中。因为合成器创建了材质和渲染，当运行时，我们没有直接的权限去控制渲染的四边形纹理，所以我们需要改变如片段着色器的属性一类的材质的权限。为了仍然可以使用他们，Ogre 3D 提供了一从监听器本身继承的监听接口。当具体的四边形产生时，这个覆写的监听器被调用。这个函数获取他生成的 pass 的 ID 和一个材质本身的指针。

通过材质指针，我们可以选择将会使用的技术，我们获取到 pass 和它所附带的片段着色器的参数。只要我们有了参数，我们就可以改变像素数目。这是一个相当长的调用式，因

为我们想要的参数是在类继承关系的底部。补充说明一下，我们可以定义参数，也可以在程序中使用合成器一类的材质来改变他们；我们甚至可以不需要监听器，因为当使用实体时，我们可以直接获取到实体的一个材质。

8.9 当程序运行的时候修改像素的数量

我们已经在程序代码中实现了改变像素的数量；那让我们走的更远一些并让我们可以通过用户输入改变像素数目。

实践时刻 —— 通过用户的输入来修改像素的数量

我们将会用到一些知识，比如用户输入和第三章(摄像机，灯光，阴影)讲的帧监听：

1. 我们的程序需要 FrameListener。添加一个新的 private 变量来存储这个指针：

```
Ogre::FrameListener* FrameListener;
```

2. 同样的，FrameListener 也应该初始化为 NULL：

```
Example79()
{
   FrameListener = NULL;
   compListener = NULL;
}
```

3. 它也应该以同样的方式销毁：

```
~Example79()
{
   if(compListener)
   {
     delete compListener;
   }
   if(FrameListener)
   {
     delete FrameListener;
```

```
    }
}
```

4. 添加函数并声明。程序的其他部分不用改变：

```
void createFrameListener()
{
    FrameListener = new Example79FrameListener(mWindow， compListener);
    mRoot->addFrameListener(FrameListener);
}
```

5. 在创建 FrameListener 之前，我们需要修改合成器监听。它需要一个 private 变量来存储我们场景想要的像素数量：

```
class CompositorListener1 : public Ogre::CompositorInstance::Listener
{
private:
    float number;
```

6. 在构造函数中以 125 初始化变量：

```
public:
    CompositorListener1()
    {
     number = 125.0f;
    }
```

7. 现在从 notifyMaterialSetup 改变覆写函数的名称为 notifyMaterialRender，并且使用 num 变量来替换一个既定的值以设置像素的数量：

```
void notifyMaterialRender(uint32 pass_id，  MaterialPtr &mat)
{
    mat->getBestTechnique()->getPass(pass_id)->
            getFragmentProgramParameters()->setNamedConstant("numpixels"， number);
}
```

8. 同时为 number 变量声明一个 getter and setter 函数：

```
void setNumber(float num)
{
   number = num;
}
float getNumber()
{
   return number;
}
```

9. 现在添加 FrameListener，它拥有三个私有变量——input manager，和我们已知的 keyboard 类，和一个指向合成器着色器的指针：

```
class Example79FrameListener : public Ogre::FrameListener
{
private:

   OIS::InputManager* _man;
   OIS::Keyboard* _key;
   CompositorListener1* _listener;
```

10. 在构造函数中，我们需要创建我们的输入系统并且保存合成器监听器的指针：

```
Example79FrameListener(RenderWindow* win，CompositorListener1* listener)
{
   _listener = listener;
   size_t windowHnd = 0;
   std::stringstream windowHndStr;
   win->getCustomAttribute("WINDOW"，  &windowHnd);
   windowHndStr << windowHnd;
   OIS::ParamList pl;
   pl.insert(std::make_pair(std::string("WINDOW")，windowHndStr.str()));
   _man = OIS::InputManager::createInputSystem( pl );
   _key = static_cast<OIS::Keyboard*>(
```

```
                                    _man->createInputObject( OIS::OISKeyboard，  false ));
}
```

11. 并且，如之前一样，我们需要摧毁我们创建的输入系统：

```
~Example79FrameListener()
{
   _man->destroyInputObject(_key);
   OIS::InputManager::destroyInputSystem(_man);
}
```

12. 覆写 *frameStarted()* 方法，在这个函数中，如果用户按下 *Escape* 键时，获取键盘的输入并关闭程序：

```
bool frameStarted(const Ogre::FrameEvent    &evt)
{
   _key->capture();

   if(_key->isKeyDown(OIS::KC_ESCAPE))
   {
    return false;
   }
```

13. 如果用户按下了上方向键，获取我们现在使用的像素总数值并且以 1 增加它。然后设置并打印这个新的值：

```
if(_key->isKeyDown(OIS::KC_UP))
{
   float num = _listener->getNumber();
   num++;
   _listener->setNumber(num);
   std::cout << num << std::endl;
}
```

14. 完成对应的步骤如果下方向键按下的话：

```
if(_key->isKeyDown(OIS::KC_DOWN))
{
   float num = _listener->getNumber();
   num--;
   _listener->setNumber(num);
   std::cout << num << std::endl;
}
```

15. 这就是全部了。现在关闭 frame started 函数：

```
   return true;
}
```

16. 编译运行程序。同时尝试不同像素数目的效果。

刚刚发生了什么？

我们扩展程序让它可以通过场景用户使用方向键来控制像素数量。第一步和第四步中添加和创建了帧监听。在第二步初始化 FrameListener 和 CompositorListener 为 NULL，并且在第三步是负责摧毁这两个指针的。在第五和第六步插入了一个新的变量到合成器监听器中，这个变量存储了想要我们的场景有的像素数目。

在第四步，我们修改之前覆写的 notifyMaterialSetup 函数为 notifyMaterialRender。这一步十分有必要，因为 notifyMaterialSetup 只在材质被创建后才会调用，但是

notifyMaterialRender 会在每次材质得到渲染时被调用。因为想要在运行时改变像素的数量，我们需要在每次画图函数调用之前修改像素的数量。当然，一个更好的办法是当像素的数量改变的时候才修改参数。这将会更节省 CPU 时间，但是我们在这里例子中不需要考虑那么多。

第八步为像素数量声明了一个 the getter and setter 方法，并且在第九步开始声明帧监听。我们需要合成器监听可以修改像素数量的值，因此我们添加一个 private 指针变量来存储它到帧监听器。

在第十步，获取了 CompositorListener 指针并在变量中存储它并在输入系统中初始化它，这部分知识我们之前已经讲过。在第十一步我们没有做什么新的事情。在十三和第十四步中使用了 getter and setter 来在合成器访问函数中来操作像素数目。在最后，第十五步，完成了帧监听并且这就是所有我们需要做的了。

动手试试 —— 减少参数的改变

改变程序使它仅当像素的数目改变时，材质中的参数才设置为一个新的值。而且不要使用 notifyMaterialRender；换而使用 notifyMaterialSetup。

8.10 添加一个分屏

目前位置，我们已经看到过如何添加一个合成器到视口，但是视口还可以做很多别的有趣的事情，比如创建一个分屏。

实践时刻 —— 添加一个分屏

在做了太多有关像素的话题之后，我们现在将会添加一个分屏

1. 我们不需要上一例子的所有代码。所以删除合成器监听器和帧监听器。

2. 我们需要第二个摄像机，所以创建一个指针来保存它

```
private:
   Ogre::Camera* mCamera2;
```

3. *createScene()* 函数仅需要创建一个 Sinbad.mesh 的实例并关联它到场景结点：

```
void createScene()
{
```

```
    Ogre::SceneNode* node = mSceneMgr->getRootSceneNode()->createChildSceneNode();
    Ogre::Entity* ent = mSceneMgr->createEntity("Sinbad.mesh");
    node->attachObject(ent);
}
```

4. 现在我们需要一个 *createCamera()* 函数，这个函数来创建一个摄像机从(0，10，20)朝向(0，0，0)：

```
void createCamera()
{
    mCamera = mSceneMgr->createCamera("MyCamera1");
    mCamera->setPosition(0，10，20);
    mCamera->lookAt(0，0，0);
    mCamera->setNearClipDistance(5);
```

5. 现在使用新的摄像机指针来存储另一个摄像机，虽和上一个摄像机看向同一点，但是这个是从位置(20，10，0)来看：

```
    mCamera2 = mSceneMgr->createCamera("MyCamera2");
    mCamera2->setPosition(20，10，0);
    mCamera2->lookAt(0，0，0);
    mCamera2->setNearClipDistance(5);
}
```

6. 我们有摄像机，但是我们现在不需要视口，所以覆写 *createViewport()* 方法：

```
void createViewports()
{
```

7. 使用第一个摄像机创建一个视口来覆盖渲染屏幕的左半部分：

```
Ogre::Viewport* vp = mWindow->addViewport(mCamera，0，0.0，0.0，0.5，1.0);
vp->setBackgroundColour(ColourValue(0.0f，0.0f，0.0f));
```

8. 然后使用第二个摄像机创建第二个视口来覆盖渲染屏幕的右半部分：

```
Ogre::Viewport* vp2 = mWindow->addViewport(mCamera2，1，0.5，0.0，0.5，1.0);
vp2->setBackgroundColour(ColourValue(0.0f，0.0f，0.0f));
```

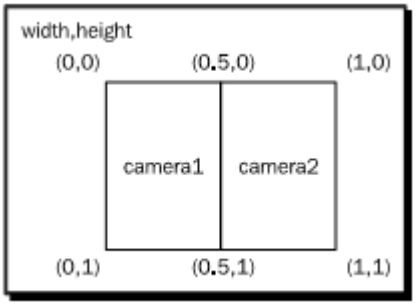
9. 两个摄像机需要正确的纵横比；否则图像将会看起来很奇怪：

```
mCamera->setAspectRatio(Real(vp->getActualWidth()) / Real(vp->getActualHeight()));
mCamera2->setAspectRatio(Real(vp2->getActualWidth()) / Real(vp2->getActualHeight()));
```

10. 编译运行程序。你将会从两个不同的视角看到相同的实例。

刚刚发生了什么？

我们给程序创建了两个视口；每个视口都有一个摄像机从不同的地方来观察我们的模型实例。因为我们想要从不同的地方观察模型，每个视口都需要自己的摄像机。因此，我们在第二步创建了一个新指针来保存我们第二个摄像机。在第三步仅创建了一个包含一个模型的简单场景。在第四步重写了 createCamera() 函数并且创建了我们第一个摄像机，这个摄像机在位置(0，10，20)朝向(0，0，0)。这意味着这个摄像机朝向沿着 Z 轴，对于例子来说，朝向模型的身前。在第五步在(20，10，0)处创建了一个摄像机，这个摄像机朝向沿着 X 轴。在第六步覆写了 createViewports()函数，这个函数在稍后的步骤中才会插入代码。在第七步创建了第一个视口，并添加第一个摄像机到 RenderWindow。这是通过 addViewport()函数完成的。此函数的第一个参数是传递显示的图像的摄像机。第二个参数是哪个视口有更高的优先级，当视口重叠的时候这个就会起作用。第三个和第四个参数是定义视口的开始点，并且第五和第六个参数是定义高度和宽度，每个参数的范围为 0 到 1.下面的图片显示了我们的渲染窗口的视口是如何建立的。

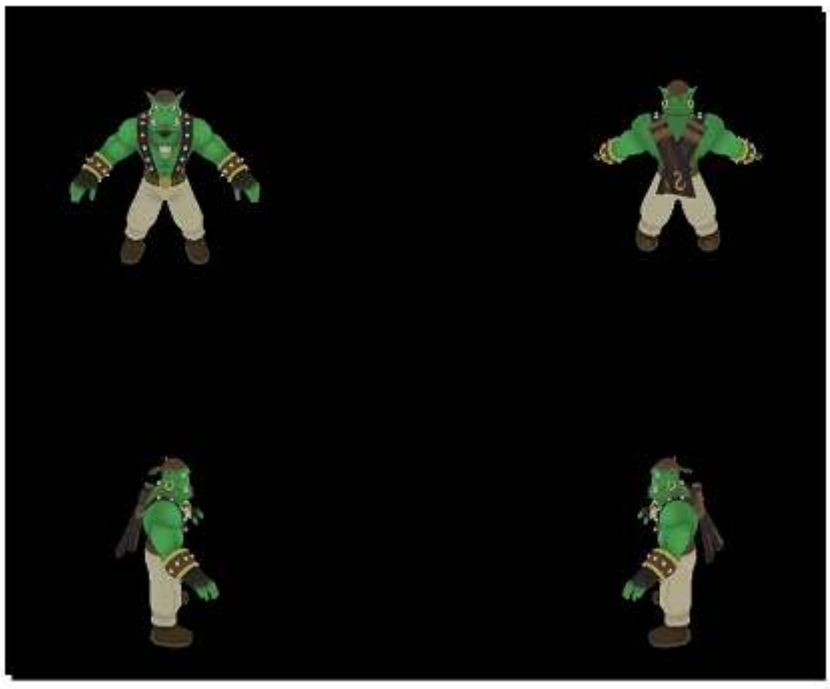


第九步仅设置了每个摄像机的纵横比，通过使用视口得到的宽度和高度信息。

同样，如果我们尝试用鼠标和键盘移动摄像机，我们可能会注意到我们仅可以控制左边的视口。这是因为只有摄像机的指针 *mCamera* 是由默认的帧监听来控制的。如果我们想要控制所有的摄像机，我们需要修改 *ExampleFrameListener*。

动手试试 —— 用视口做更多的事

创建一个有四个视口的程序 —— 一个是看前面，一个看后面，一个看左边，一个看右边。结果应该看起来如下图：



实践时刻 —— 选择一个颜色通道

我们将会使用之前的代码，但是我们将会对其进行一些删改：

1. 首先创建一个片段着色器。除了正常的参数，添加一个 float4 类型的 uniform 参数来存储颜色通道的因子。使用这些因子来乘以颜色，我们将会从原始的场景纹理中检索颜色：

```
void MyFragmentShader10(float2 uv        : TEXCOORD0，
        out float4 color : COLOR，
        uniform sampler2D texture，
        uniform float4 factors)
{
   color = tex2D(texture，  uv);
   color *= factors;
}
```

2. 使用片段着色器来创建一个新的材质并且用默认的值(1，1，1，0)来添加。这意味着如果没有改变的参数，场景将会正常渲染：

```
fragment_program MyFragmentShader10 cg
{
   source Ogre3DBeginnerGuideShaders.cg
   entry_point MyFragmentShader10
   profiles ps_1_1    arbfp1

   default_params
   {
    param_named factors float4 1 1 1 0
   }
}
material Ogre3DBeginnersGuide/Comp7
{
   technique
      {
         pass
```

```
            {
        fragment_program_ref MyFragmentShader10
                    {


                    }

        texture_unit
                    {
                    }
            }
        }
}
```

3. 然后使用这个材质添加一个合成器：

```
compositor Compositor9
{
    technique
     {
            texture scene target_width target_height PF_R8G8B8
            target scene
            {
                input previous
            }
            target_output
            {
                input none
                pass render_quad
                {
                    material Ogre3DBeginnersGuide/Comp7
                    input 0 scene
                }
            }
```

```
    }
}
```

4. 我们有三个颜色通道，所以我们将会需要三个合成器监听来相应的改变参数。首先，添加一个红色通道的监听。仅设置颜色因子在材质建立时并且我们不需要在运行时改变它们：

```
class CompositorListener2 : public Ogre::CompositorInstance::Listener
{
public:
  void notifyMaterialSetup (uint32 pass_id，  MaterialPtr &mat)
  {
    mat->getBestTechnique()->getPass(pass_id)->getFragmentProgramParameters()
                                ->setNamedConstant("factors"，Ogre::Vector3(1，0，0));
  }
}
```

5. 现在，添加绿色和和蓝色颜色通道：

```
class CompositorListener3 : public Ogre::CompositorInstance::Listener
{
public:
  void notifyMaterialSetup (uint32 pass_id，  MaterialPtr &mat)
  {
   mat->getBestTechnique()->getPass(pass_id)->
            getFragmentProgramParameters()->setNamedConstant(
            "factors"，Ogre::Vector3(0，1，0));
  }
};

class CompositorListener4 : public Ogre::CompositorInstance::Listener
{
public:
  void notifyMaterialSetup (uint32 pass_id，  MaterialPtr &mat)
```

```
    {
      mat->getBestTechnique()->getPass(pass_id)->getFragmentProgramParameters()
                                        ->setNamedConstant("factors"，Ogre::Vector3(0，0，1));
    }
};
```

6. 替换摄像机指针，添加四个视口到程序中：

```
class Example83 : public ExampleApplication
{
private:
    Ogre::Viewport* vp;
    Ogre::Viewport* vp2;
    Ogre::Viewport* vp3;
    Ogre::Viewport* vp4;
```

7. 创建一个我们将会使用的摄像机并且把他放到可以看到 Sinbad 前面的位置：

```
void createCamera()
{
    mCamera = mSceneMgr->createCamera("MyCamera1");
    mCamera->setPosition(0，10，20);
    mCamera->lookAt(0，0，0);
    mCamera->setNearClipDistance(5);
}
```

8. 修改 createViewport() 函数仅使用一个摄像机并为两个新的视口添加必要的代码：

```
void createViewports()
{
    vp = mWindow->addViewport(mCamera，0，0.0，0.0，0.5，0.5);
    vp->setBackgroundColour(ColourValue(0.0f，0.0f，0.0f));
    vp2 = mWindow->addViewport(mCamera，1，0.5，0.0，0.5，0.5);
    vp2->setBackgroundColour(ColourValue(0.0f，0.0f，0.0f));
```

```
    vp3 = mWindow->addViewport(mCamera，2，0.0，0.5，0.5，0.5);
    vp3->setBackgroundColour(ColourValue(0.0f，0.0f，0.0f));
    vp4 = mWindow->addViewport(mCamera，3，0.5，0.5，0.5，0.5);
    vp4->setBackgroundColour(ColourValue(0.0f，0.0f，0.0f));
    mCamera->setAspectRatio(Real(vp->getActualWidth()) / Real(vp->getActualHeight()));
}
```

9. 添加三个指针来存储我们上面创建的合成器监听：

```
CompositorListener2* compListener;
CompositorListener3* compListener2;
CompositorListener4* compListener3;
```

10. 在构造函数中初始化每个为 NULL：

```
Example83()
{
    compListener = NULL;
    compListener2 = NULL;
    compListener3 = NULL;
}
```

11. 并且，同样，在析构函数中删除他们：

```
~Example83()
{
     if(compListener)
    {
     delete compListener;
    }
    if(compListener2)
    {
     delete compListener2;
    }
```

```
    if(compListener3)
    {
      delete compListener3;
    }
}
```

12. 在 *createScene()* 函数中，在创建模型实例和场景结点之后，添加必要的代码以添加合成器到我们第一个视口，激活它，并关联它到合成器监听，这个监听仅红色被渲染：

```
Ogre::CompositorManager::getSingleton().addCompositor(vp， "Compositor9");
Ogre::CompositorManager::getSingleton().
                                    setCompositorEnabled(vp， "Compositor9"， true);
Ogre::CompositorInstance*comp=  Ogre::CompositorManager::getSingleton().
                                getCompositorChain(vp)->getCompositor("Compositor9");
compListener = new CompositorListener2();
comp->addListener(compListener);
```

13. 给第二和第三个视口使用同样的办法，但是使用不同的绿色和蓝色合成器监听：

```
Ogre::CompositorManager::getSingleton().addCompositor(vp2，"Compositor9");
Ogre::CompositorManager::getSingleton().
                                    setCompositorEnabled(vp2，"Compositor9"， true);
Ogre::CompositorInstance* comp2 = Ogre::CompositorManager::getSingleton().
                                getCompositorChain(vp2)->getCompositor("Compositor9");
compListener2 = new CompositorListener3();
comp2->addListener(compListener2);

Ogre::CompositorManager::getSingleton().addCompositor(vp3， "Compositor9");
Ogre::CompositorManager::getSingleton().
                                    setCompositorEnabled(vp3， "Compositor9"， true);
Ogre::CompositorInstance*comp3= Ogre::CompositorManager::getSingleton().
                                getCompositorChain(vp3)->getCompositor("Compositor9");
compListener3 = new CompositorListener4();
comp3->addListener(compListener3);
```

14. 现在编译运行程序。你将会看到四个完全相似的图像，仅仅是渲染的颜色通道有区别。在左上角的那个，仅仅是红色通道可视；在右上角那个，仅仅是绿色可是；在左下角的那个，是蓝色的；在右下角的那个，是有所有颜色通道的：

刚刚发生了什么？

我们把这章例子的知识综合到一起创建了一个应用程序，它使用四个视口和一个结合了三个合成器监听合成器以观察他们每个本身的颜色通道和结合后的效果。这个例子里其实没有什么新的不同；如果需要，查阅其他的例子以理解这个例子。 】

8.11 小结

这章我们接触到很多有关合成器和视口的知识。

具体来说，我们学习了：

1. 如何创建合成器脚本并且如何添加他们到我们的场景。
2. 如何使用合成器和片段着色器来操作我们的场景。
3. 着色器的参数和如何在材质脚本中改变他们的值或者直接在程序代码中修改。
4. 如何结合合成器来防止我们多次的重写代码。
5. 结合我们之前所学来创建了一个可以由用户控制的合成器。

我们现在已经学习了很多有关 Ogre 3D 的知识；仅有一个非常重要的话题还没有涉及。目前位置，我们一直依赖着 *ExampleApplication*。在下一章，我们将会写我们自己的 *ExampleApplication*。

第九章 Ogre 3D 的启动顺序

我们在学习这本书的过程中了解了很多基础的知识。这一章将会学习一个尚未了解的主题：如何不依赖 ExampleApplication 创建我们自己的程序。当我们学习完这个主题，这章会重复一些之前章节学过的专题，使用一个新的程序的类来做各种示例。

这一章，我们将会：

1. 学习如何启动我们的 Ogre 3D。
2. 解析 resources.cfg 文件来加载我们需要的模型。
3. 结合我们之前章节所学来创建一个小的示例程序来展示我们之前所学。

那么我们开始吧….

9.1 启动 Ogre 3D

目前为止，*ExampleApplication* 类已经为我们启动和初始化了 Ogre 3D；现在我们将会自己来实现这个过程。

实践时刻 —— 启动 Ogre 3D

这次我们将会在一个空白的文件上开始。

1. 用一个空白的代码文件作为开始，引入 Ogre3d.h 头文件，并创建一个空的主函数：

```
#include "Ogre\Ogre.h"
int main (void)
{
   return 0;
}
```

2. 创建一个 Ogre 3D Root 类的实例；这个类需要 plugin.cfg 作为参数：

```
Ogre::Root* root = new Ogre::Root("plugins_d.cfg");
```

3. 如果配置对话框不能使用或者用户选择退出，关闭应用程序：

```
if(!root->showConfigDialog())
```

```
{
    return -1;
}
```

4. 创建一个渲染窗口：

```
Ogre::RenderWindow* window = root->initialise(true， "Ogre3D Beginners Guide");
```

5. 下一步创建一个新的场景管理器：

```
Ogre::SceneManager* sceneManager = root->createSceneManager(Ogre::ST_GENERIC);
```

6. 创建一个摄像机并给其命名：

```
Ogre::Camera* camera = sceneManager->createCamera("Camera");
camera->setPosition(Ogre::Vector3(0， 0， 50));
camera->lookAt(Ogre::Vector3(0， 0， 0));
camera->setNearClipDistance(5);
```

7. 使用这个摄像机，创建一个视口并设置背景颜色为黑色：

```
Ogre::Viewport* viewport = window->addViewport(camera);
viewport->setBackgroundColour(Ogre::ColourValue(0.0， 0.0， 0.0));
```

8. 现在，使用视口并设置摄像机的纵横比：

```
camera->setAspectRatio(Ogre::Real(viewport->getActualWidth())/
Ogre::Real(viewport->getActualHeight()));
```

9. 最后，告诉 root 开始渲染：

```
root->startRendering();
```

10. 编译运行程序；你将会看到正常的配置对话框然后是黑色的窗口。这个窗口不能按 Escape 来关闭，因为我们目前没有添加按键控制。你可以在启动程序的控制台中按 CTRL+C 来关闭程序。

刚刚发生了什么？

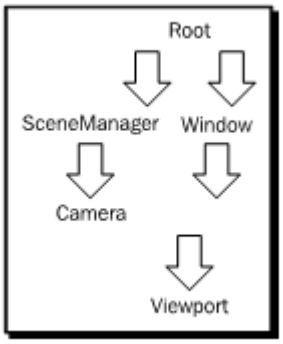
我们第一次没有在 ExampleApplication 帮助下创建了 Ogre 3D 程序。因为我们不再使用 ExampleApplication 了，需要引入 Ogre3D.h ，它之前被事先在引入在 ExampleApplication.h 中。在我们使用 Ogre 3D 之前，我们需要一个 root 实例。这个 root 是一个可以管理 Ogre 3D 高层的类，有着创建和保存其他对象的工厂函数，加载和卸载需要的插件和做其他的一些东西。我们给 root 实例一个参数：定义加载插件的文件。

```
Root(const String & pluginFileName = "plugins.cfg"，const String & configFileName =
"ogre.cfg"，const String & logFileName = "Ogre.log")
```

除了插件配置文件的名称，函数还需要 Ogre 配置文件的名称和日志文件。我们需要改变第一个文件的名称，因为我们使用的是 debug 版的程序，因此我们想要加载 debug 插件。默认的文件名为 plugins.cfg，是 Ogre 3D SDK release 文件夹下的文件名，但是我们程序是在 debug 文件夹运行，所以文件名称为 plugins_d.cfg。

ogre.cfg 包含了我们在 Ogre 程序开始时在配置对话框中选择的选项。这是方便了用户每次可以以相同的配置启动程序。通过这个文件，Ogre 3D 可以记住他的选项并使用它们作为下次启动的选项。如果文件不存在，文件将会自动创建，所以我们不需要添加_d 到这个文件名后面，可以使用默认的参数；这种方式对于日志文件也是相同的。

使用 root 实例，第三步我们让 Ogre 3D 给用户显示了配置对话框。当用户退出对话框或发生了什么错误，我们将会返回-1 并让程序关闭。否则，我们会执行第四步的创建一个新的渲染窗口和一个新的场景管理器。使用场景管理器，我们创建了一个摄像机，使用摄像机我们创建了视口；然后使用视口，我们计算了摄像机的纵横比。摄像机和视口的创建不是什么新的知识了；我们在第三章(摄像机，灯光和阴影)已经学习过它们。在创建所有的需求之后，我们告诉根实例来开始渲染，这样我们的结果就可视了。下面的图示显示了创建别的对象需要什么对象：



9.2 添加资源

我们现在创建了我们第一个不需要 ExampleApplication 类的 Ogre 3D 程序。但是一个重要的东西遗漏了：我们目前还未加载和渲染一个模型。】

实践时刻 —— 加载 Sinbad 的 mesh

我们已经有了基本的程序，现在让我们添加一个模型。 】

1. 在设置完纵横比和开始渲染之前，添加包含 Sinbad 模型的 zip 压缩文件到我们的资源：

```
Ogre::ResourceGroupManager::getSingleton().
                                addResourceLocation("../../Media/packs/Sinbad.zip"， "Zip");
```

2. 我们现在不想要索引更多的资源，所以现在仅索引添加的资源：

```
Ogre::ResourceGroupManager::getSingleton().initialiseAllResourceGroups();
```

3. 现在创建一个 Sinbad mesh 的实例并添加只场景：

```
Ogre::Entity* ent = sceneManager->createEntity("Sinbad.mesh");
sceneManager->getRootSceneNode()->attachObject(ent);
```

4. 编译运行程序；你应看到 Sinbad 在屏幕正中央：

刚刚发生了什么？

我们使用了 ResourceGroupManager 来检索包含 Sinbad mesh 和纹理文件的 zip 压缩包，做完这个，我们第三步告诉程序用 createEntity()来加载数据。

9.3 使用 resources.cfg

为每一个我们想要加载的 zip 或者文件夹新添加一行代码是一个非常乏味的工作，所以为避免这样。ExampleApplication 类使用了一个称为 resources.cfg 配置文件，这个配置文件列出了每个文件夹和压缩包，并且使用这个文件我们加载了所有内容。让我们现在重复实现一下这个功能。

实践时刻 —— 使用 resources.cfg 加载我们的模型

使用我们之前的程序代码，这里我们将会解析 resources.cfg。

1. 替换加载 zip 包的代码为一个指向 resources_d.cfg 的配置文件的实例：

```
Ogre::ConfigFile cf;
cf.load(“resources_d.cfg”);
```

2. 首先获取一可以遍历配置文件的每个区块的迭代器：

```
Ogre::ConfigFile::SectionIterator sectionIter = cf.getSectionIterator();
```

3. 定义三个字符串以保存从配置文件中提取的存数据并且遍历每个区块：

```
Ogre::String sectionName， typeName， dataname;
while (sectionIter.hasMoreElements())
{
```

4. 获取区块的名称：

```
sectionName = sectionIter.peekNextKey();
```

5. 获取区块中包含的设置，同时，提前创建一个区块的迭代器：

```
Ogre::ConfigFile::SettingsMultiMap *settings = sectionIter.getNext();
```

```
Ogre::ConfigFile::SettingsMultiMap::iterator i;
```

6. 在区块中遍历每个设置：

```
for (i = settings->begin(); i != settings->end(); ++i)
{
```

7. 使用迭代器来获取资源的名称和类型：

```
typeName = i->first;
dataname = i->second;
```

8. 使用资源名称，类型和区块的名称并添加至资源索引：

```
Ogre::ResourceGroupManager::getSingleton().
                                addResourceLocation(dataname，typeName， sectionName);
```

9. 编译运行程序，你将会看到和之前一样的场景：

刚刚发生了什么？

在第一步，我们使用了 Ogre 3D 的另一个助手类。这个类可以简单的的加载和解析简单的配置文件，这个文件由 name-value 对组成。我们把带 debug 模式字尾的文件名写入程序；这并不是好的习惯并且在实际的程序中我们将会使用#ifdef 来依据 debug 和 release 模式以改变资源文件的文件名。ExampleApplication 类就是这样做的；让我们看下 ExampleApplication.h 第 384 行：

```
#if OGRE_DEBUG_MODE
    cf.load(mResourcePath + "resources_d.cfg");
#else
    cf.load(mResourcePath + "resources.cfg");
#endif
```

配置文件的结构

配置文件由助手类加载，其有着一个简单的结构；这里是 resource.cfg 的例子。当然你的 resources.cfg 里面包含着不同的路径：

```
[General]
FileSystem=D:/programming/ogre/ogre_trunk_1_7/Samples/Media
```

以 [General] 作为一区段的开始，以另一个 [sectionname] 的出现作为结束。每个配置文件可以包含很多的区段；在第二步我们创建了一个迭代器来遍历文件中所有的区段，并在第三步我们使用了一个 while 循环，以遍历完每个区段作为结束

一个区段包含不同的设置并且每个设置等同于一个键和一个值。我们赋值给键 FileSystem 值为 D:/programming/ogre/ogre_trunk_1_7/Samples/ Media。在第四步，我们创建了一个新的迭代器，这样我们就可以遍历每个设置了。设置是一 name-value 数对。我们遍历这个 map ，对于每个实例我们使用 map 的键作为资源的类型，map 的数据作为路径。使用区段名作为资源组，我们在第八步使用资源组管理器来添加资源。只要我们解析完这个文件，我们就索引完毕所有的文件。

9.4 创建应用程序类

我们现在已经有自己 Ogre 3D 程序的主要部分了，但是所有代码都在 main 函数中，这并不是我们想要的那种可重用的代码。

实践时刻 —— 创建一个类

使用之前用过的代码，我们现在将会创建一个类来实现 Ogre 代码从 main 函数中的分离。

1. 创建有两个私有的指针的 MyApplication 类，一个指向 Ogre 3D SceneManager，另一个指向 Root 类：

```
class MyApplication
{
private:
   Ogre::SceneManager* _sceneManager;

Ogre::Root* _root;
```

2. 这个类的剩余部分应为 public：

```
public:
```

3. 创建一个 *loadResources()* 函数，这个函数加载 resources.cfg 配置文件：

```
void loadResources()
{
   Ogre::ConfigFile cf;
   cf.load("resources_d.cfg");
```

4. 遍历配置文件的所有区段：

```
Ogre::ConfigFile::SectionIterator sectionIter =
cf.getSectionIterator();
Ogre::String sectionName， typeName， dataname;
while (sectionIter.hasMoreElements())
{
```

5. 获取区段名称和设置的迭代器：

```
sectionName = sectionIter.peekNextKey();
Ogre::ConfigFile::SettingsMultiMap *settings = sectionIter.getNext();
Ogre::ConfigFile::SettingsMultiMap::iterator i;
```

6. 迭代所有的设置并添加每个资源：

```
      for (i = settings->begin(); i != settings->end(); ++i)
      {
         typeName = i->first;
         dataname = i->second;
         Ogre::ResourceGroupManager::getSingleton().addResourceLocation(
                                                dataname， typeName， sectionName);
      }
   }
   Ogre::ResourceGroupManager::getSingleton().
   initialiseAllResourceGroups();
}
```

7. 同样创建一个 startup()函数，使用 plugins.cfg 创建一个 Ogre 3D root 类的实例：

```
int startup()
{
_root = new Ogre::Root("plugins_d.cfg");
```

8. 显示配置窗口当用户退出时，返回-1 并关闭程序：

```
if(!_root->showConfigDialog())
{
   return -1;
}
```

9. 创建 RenderWindow 和 SceneManager：

```
Ogre::RenderWindow* window = _root->initialise(true，"Ogre3D Beginners Guide");
_sceneManager = _root->createSceneManager(Ogre::ST_GENERIC);
```

10. 创建一摄像机和一个视口：

```
Ogre::Camera* camera = _sceneManager->createCamera("Camera");
camera->setPosition(Ogre::Vector3(0，0，50));
camera->lookAt(Ogre::Vector3(0，0，0));
camera->setNearClipDistance(5);
Ogre::Viewport* viewport = window->addViewport(camera);
viewport->setBackgroundColour(Ogre::ColourValue(0.0，0.0，0.0));
camera->setAspectRatio(Ogre::Real(viewport->getActualWidth())/
Ogre::Real(viewport->getActualHeight()));
```

11. 调用函数来加载我们的资源并且然后函数来创建创建一个场景；最后，Ogre 3D 可以开始渲染了：

```
loadResources();
createScene();
_root->startRendering();
return 0;
```

12. 然后创建包含创建 SceneNode 和 Entity 的 *createScene()* 函数：

```
void createScene()
{
   Ogre::Entity*ent=_sceneManager->createEntity("Sinbad.mesh");
   _sceneManager->getRootSceneNode()->attachObject(ent);
}
```

13. 我们需要构造函数设置两个指针为 NULL，即使它不被赋值，这样我们也可以删除它：

```
MyApplication()
{
   _sceneManager = NULL;
_root = NULL;
}
```

14. 当实例摧毁的时候，我们需要删除 root 实例，所以声明一个析构函数来处理：

```
~MyApplication()
{
   delete _root;
}
```

15. 最后剩下的事情是修改 main 函数：

```
int main (void)
{
   MyApplication app;
   app.startup();
   return 0;
}
```

16. 编译运行程序；场景依旧没有改变。

刚刚发生了什么？

我们重构了开始的代码这样不同的函数得到了更好的组织。我们也添加了一个析构函数这样当程序关闭的时候创建的实例可以得到删除。一个问题就是我们的析构函数不能够得到调用，因为 startup()函数永远不返回，这样就没有什么方法来关闭我们的程序。我们需要添加一个 FrameListener 来告诉 Ogre 3D 停止渲染。

9.5 添加一个帧监听

我们之前已经使用过 ExampleFrameListener；这一次我们将会使用我们自己声明的接口。

实践时刻 —— 添加一个帧监听

使用之前的代码，我们将会添加我们自己的 FrameListener 声明

1. 创建一个新的称为 MyFrameListener，并列出三个公有的事件响应函数：

```
class MyFrameListener : public Ogre::FrameListener
{
public:
```

2. 首先，声明 frameStarted 函数，现在这个函数以返回 false 来结束函数：

```
bool frameStarted(const Ogre::FrameEvent& evt)
{
    return false;
}
```

3. 同样我们需要一个 frameEnded 函数，也是返回 false：

```
bool frameEnded(const Ogre::FrameEvent& evt)
{
    return false;
}
```

4. 最后我们声明的一个函数是 *frameRenderingQueued* 函数，也是返回 false：

```
bool frameRenderingQueued(const Ogre::FrameEvent& evt)
{
   return false;
}
```

5. 类的主体需要一个指针来存储 FrameListener：

```
MyFrameListener* _listener;
```

6. 切记在构造函数用设置监听初始化的值为 NULL：

```
_listener = NULL;
```

7. 让析构函数删除这个实例：

```
delete _listener;
```

8. 最后创建一个 FrameListener 的实例，并添加它到 root 对象；这些代码在 startup()函数中：

```
_listener = new MyFrameListener();
_root->addFrameListener(_listener);
```

9. 编译运行程序；程序运行会直接关闭。

刚刚发生了什么？

我们创建自己的 FrameListener 类，这个类不依赖于 ExampleFrameListener 的实现。这次我们直接从 FrameListener 接口继承。这个接口包含了三个虚函数，这三个虚函数需要我们在自己的 FrameListener 中覆写。我们已经知道 frameStarted()函数，但是其他两个是新的函数。所有三个函数返回 false，都是通知 Ogre 3D 停止渲染并且关闭程序。使用我们的实现，添加一个 FrameListener 到 root 实例并且开始渲染程序；没什么特别的，这里程序直接关闭。

9.6 研究 FrameListener 的功能

我们的 FrameListener 实现有三个函数；每个都是在不同的时刻调用。我们将会研究一下他们是在渲染的哪个过程被调用的。

实践时刻 —— 用 FrameListener 的实现来做实验

当 FrameListener 调用时，利用控制台的输出检查哪里被调用。

1. 首先当每个函数调用时，使其输出一个消息到控制台。

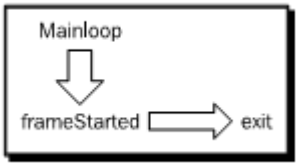
```
bool frameStarted(const Ogre::FrameEvent& evt)
{
   std::cout<<"Frame started"<<std::endl;
   return false;
}

bool frameEnded(const Ogre::FrameEvent& evt)
{
   std::cout<<"Frame ended"<<std::endl;
   return false;
}
bool frameRenderingQueued(const Ogre::FrameEvent& evt)
{
   std::cout<<"Frame queued"<<endl;
   return false;
}
```

2. 编译运行程序；在控制台中你会看到第一个字符串 Frame started。

刚刚发生了什么？

我们添加了一个 debug 消息输出到每个 FrameListener 的函数里面以观察哪个函数被调用了。运行程序，我们注意到只有第一个 debug 信息输出了。这是因为 frameStarted 返回了 false，这是一个请求 root 实例关闭程序的信号。



既然我们知道当 *frameStarted()* 返回 false，那让我们看一下当 *frameStarted()* 返回 true 的时候发生了什么。

实践时刻 —— 在 frameStarted 函数中返回 true

现在我们将会修改 FrameListener 的行为以研究，这个修改是如何影响了其行为。

1. 改变 frameStarted 并返回 true：

```
bool frameStarted(const Ogre::FrameEvent& evt)
{
   std::cout<<"Frame started"<<endl;
   return true;
}
```

2. 编译运行程序。在程序关闭之前，你将会看到渲染的场景闪了一下并且在控制台输出里会有下面两行：

Frame started

Frame queued

刚刚发生了什么？

现在，frameStarted 函数返回 true 并且这使得 Ogre 3D 继续渲染，直到 frameRenderingQueued 返回 false 为止。我们这次在 frameRenderingQueued 函数调用的时候看了一个场景，渲染的缓存在程序关闭之前得到了交换。 】

9.7 双缓存

当一个场景被渲染的时候，它并不是直接渲染到会直接显示到显示设备中缓存中。正常来说，场景被渲染到第二缓存，并且当渲染结束的时候，缓存得到了交换。这样可以避免如果在渲染同一缓存时，一些尚未处理完成的效果就会显示在显示屏上。FrameListener 的

frameRenderingQueued 函数，当场景渲染转至未显示的后台缓存后被调用。在缓存交换之前，渲染已经完毕但尚未显示。直接调用完 frameRenderingQueued，缓存得到交换，然后程序得到 return 的值，最后关闭程序。这就是我们只看到一眼图片。现在，我们将会研究当 frameRenderingQueued 返回 true 后会有什么发生。

实践时刻 —— 在 frameRenderingQueued 返回 true

我们再次修改代码来测试 FrameListener 的行为。

1. 改变 frameRenderingQueued 并返回 true：

```
bool frameRenderingQueued (const Ogre::FrameEvent& evt)
{
   std::cout<<"Frame queued"<<endl;
   return true;
}
```

2. 编译运行程序。你将会在程序关闭之前看到 Sinbad 一小会，下面这三行是控制台输出的：

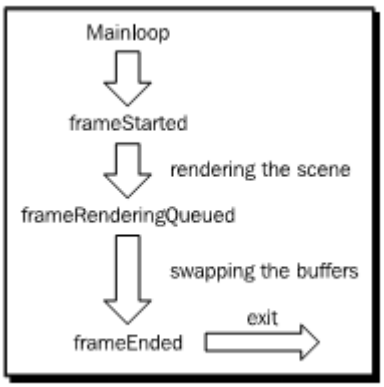
Frame started

Frame queued

Frame ended

刚刚发生了什么？

既然 frameRenderingQueued 函数返回为 true，那么 Ogre 3D 将会继续渲染直到 frameEnded 方法返回 false。



如我们上一个例子，渲染的缓存交换了，这样我们可以看到短时间的场景。当帧被渲染后，frameEnded 将会返回 false，这将会关闭程序，在这种情况下，不需要改变对 FrameListener 的理解。

实践时刻 —— 在 frameEnded 函数中返回 true

现在让我们测试最后一种可能性：

1. 改变 frameEnded(译注* 原文错为 frameRenderingQueued)，让它返回 true：

```
bool frameEnded (const Ogre::FrameEvent& evt)
{
   std::cout<<"Frame ended"<<endl;
   return true;
}
```

2. 编译运行程序。你将会看到 Sinbad 的实例，并且下面的三行输出会一直循环：

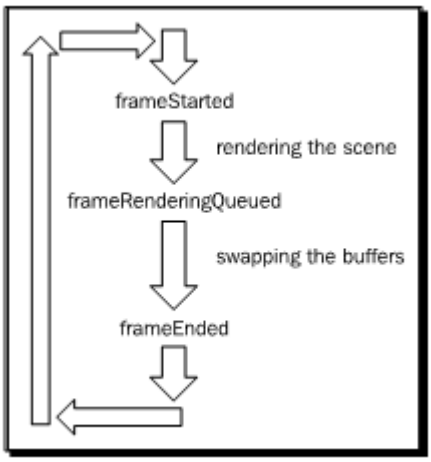
Frame started

Frame queued

Frame ended

刚刚发生了什么 ？

现在，所有的事件处理函数都是返回 true，并且，除非我们自己关闭程序，程序永不会直接关闭。



9.8 添加输入

我们现在有了一直运行的程序并且必须强制关闭；而这并不是好的方式。那么让我们添加输入设置并且使得程序可以按下 Escape 退出。

实践时刻 —— 添加输入

既然我们已经知道 FrameListener 是如何工作的了，那么让我们添加一些输入吧。

1. 我们需要引入 OIS 头文件来使用 OIS：

```
#include "OIS\OIS.h"
```

2. 删除 FrameListener 中的所有函数，并添加两个私有成员来存储 InputManager 和 Keyboard：

```
OIS::InputManager* _InputManager;
OIS::Keyboard* _Keyboard;
```

3. FramListener 需要一个指向 RenderWindow 类的指针来初始化 OIS，所以我们需要一个以窗口指针作为参数的构造函数：

```
MyFrameListener(Ogre::RenderWindow* win)
{
```

4. OIS 使用参数列表来初始化，我们同样需要一个字符串形式的窗口句柄来构建参数列

表；创建三个必要的变量来保存数据：

```
OIS::ParamList parameters;
unsigned int windowHandle = 0;
std::ostringstream windowHandleString;
```

5. 获得 RenderWindow 的句柄并转换它为一个字符串：

```
win->getCustomAttribute("WINDOW"， &windowHandle);
windowHandleString << windowHandle;
```

6. 使用键值”WINDOW”来添加字符串类型的句柄到参数表

```
parameters.insert(std::make_pair("WINDOW"， windowHandleString.str()));
```

7. 使用参数列表来创建 InputManager:

```
_InputManager = OIS::InputManager::createInputSystem(parameters);
```

8. 用管理器来创建 keyboard：

```
_Keyboard = static_cast<OIS::Keyboard*>(
                    _InputManager->createInputObject( OIS::OISKeyboard， false ));
```

9. 还有，记着我们构造函数中创建的，我们需要在析构函数在把他摧毁：

```
~MyFrameListener()
{
  _InputManager->destroyInputObject(_Keyboard);
  OIS::InputManager::destroyInputSystem(_InputManager);
}
```

10. 创建一个新的 frameStarted 函数，来获取当前的键盘状态，并且如果按下了 Escape，将会返回 false;否则，他会返回 true:

```
bool frameStarted(const Ogre::FrameEvent& evt)
{
```

```
    _Keyboard->capture();
    if(_Keyboard->isKeyDown(OIS::KC_ESCAPE))
    {
        return false;
    }
    return true;
}
```

11. 最后要做的一件事就是使用一个指针来改变 FrameListener 的实例，这样在 startup 函数中就可以渲染窗口了：

```
_listener = new MyFrameListener(window);
_root->addFrameListener(_listener);
```

12. 编译运行程序。你会看到场景，并且现在可以按下 Escape 键来关闭程序。

刚刚发生了什么？

我们用之前第四章(获得用户输入和使用帧监听)讲过的方式给 FramListener 添加了输入的能力。但是这次有点不同的是，我们没有使用任何 example 中给的类，这次是我们自己写出来的版本。

小测试 —— 三个事件处理

哪三个函数提供了 FrameListener 的接口，这三个函数是在什么时候被调用到？

9.9 我们自己的主循环

我们已经试了了 startRenering 函数来启动我们的程序。但在这之后，依靠 FrameListener 是我们目前唯一渲染帧的方式。但是一些时候我们放弃渲染的主循环是不太可能或者不是我们期望的；为了解决这个情况，Ogre 3D 提供了另一个方式，不需要我们去放弃渲染的主循环

时间时刻 —— 使用我们自己的渲染循环

使用我们之前的代码，现在我们将会使用我们的渲染循环。

1. 我们的程序需要知道是否要一直循环下去；所以添加一个布尔类型的私有成员变量

以记录成员的状态：

```
bool _keepRunning;
```

2. 删除在 startup 函数中的 startRendering 函数。

3. 添加一个新的称为 renderOneFrame 的函数，这个函数调用了 root 实例的 renderOneFrame 函数，并且可以把返回值保存在_ keepRunning 成员变量中。在调用这个之前，添加一个函数在处理所有的窗口事件：

```
void renderOneFrame()
{
   Ogre::WindowEventUtilities::messagePump();
   _keepRunning = _root->renderOneFrame();
}
```

4. 添加一个获取_keepRunning 成员变量的函数：

```
bool keepRunning()
{
   return _keepRunning;
}
```

5. 添加一个 while 循环到主函数中，只要 keepRunning 函数返回 true，循环将一直进行下去。在循环体里，提用程序的 renderOneFrame 函数：

```
while(app.keepRunning())
{
   app.renderOneFrame();
}
```

6. 编译并运行程序。看到的效果和上一个例子没有什么明显的不同。

刚刚发生了什么

我们把主循环的控制权从 Ogre 3D 移到我们的程序之中。在改变之前，Ogre 3D 使用了一个内部的主循环，这个主循环之前我们是无法控制的，并且不得不依赖 FrameListener 来通知我们帧是否被渲染了。

现在我们得有自己的主循环。为实现这个功能，我们需要一个 Boolean 成员变量，这个变量来通知我们程序是否想要继续渲染；这个变量我们在第一步中添加完毕。在第二步中移除了之前调用的 startRendering 函数，这样我们就不会把渲染的主循环交给 Ogre 了。在第三步，我们创建了一个函数，此函数首先调用了 Ogre 3D 的一个帮助函数，这个函数来处理来自操作系统的窗口事件信息。然后他发送自从上一帧结束后所有的信息，这样程序才能在操作系统的窗口系统中表现正常。

在这之后我们调用了 Ogre 3D 的 renderOneFrame 函数，这个函数正如它的名字所暗示的那样：渲染每帧，并且调用 frameStarted ，frameRenderingQueued 和 frameEnded 在帧监听中注册的事件处理程序，如果其中的一个子函数 false，那么此函数返回 false。我们通过把返回值赋值给_ keepRunning 成员变量，这样我们就可以检测程序是否在持续运行。当 renderOneFrame 返回 true 的时候，程序应该改运行下去；如果返回 false，我们知道是帧监听想要关闭程序，这样我们设置_keepRunning 变量为 false。第四步仅添加了 getter 函数来获取_keepRunning 的值。

第五步，我们使用了_keepRunning 变量作为循环的条件。这意味着如果_keepRunning 返回 true，他将持续运行下去，直到一个 FrameListener 返回 false，这将会导致 while 循环退出然后整个程序关闭。在 while 循环内我们调用程序的 renderOneFrame 函数以最新的渲染信息更新渲染窗口。这就是我们需要创建自己主循环的必要条件。

9.10 添加一个摄像机(再次)

实践时刻 —— 添加一个帧监听

使用我们自己的帧监听，我们将会添加用户控制的摄像机。

1. 为控制摄像机我们需要一个鼠标的指针，一个摄像机的指针和一个定义摄像机移动速度的的变量：

```
OIS::Mouse* _Mouse;
Ogre::Camera* _Cam;
float _movementspeed;
```

2. 修改构造函数并添加摄像机指针作为一个新的参数并设置移动的速度为 50：

```
MyFrameListener(Ogre::RenderWindow* win，Ogre::Camera* cam)
{
  _Cam = cam;
  _movementspeed = 50.0f;
```

3. 使用 InputManager 初始话 mouse 指针：

```
_Mouse = static_cast<OIS::Mouse*>(_InputManager
                                  ->createInputObject( OIS::OISMouse，  false ));
```

4. 并记住在析构函数里面摧毁它：

```
_InputManager->destroyInputObject(_Mouse);
```

5. 添加 WASD 键来移动摄像机的代码和 frameStarted 事件处理中函数来控制摄像机移动速度的代码：

```
Ogre::Vector3 translate(0，0，0);
if(_Keyboard->isKeyDown(OIS::KC_W))
{
   translate += Ogre::Vector3(0，0，-1);
}
if(_Keyboard->isKeyDown(OIS::KC_S))
{
   translate += Ogre::Vector3(0，0，1);
}
if(_Keyboard->isKeyDown(OIS::KC_A))
{
   translate += Ogre::Vector3(-1，0，0);
}
if(_Keyboard->isKeyDown(OIS::KC_D))
{
   translate += Ogre::Vector3(1，0，0);
}
_Cam->moveRelative(translate*evt.timeSinceLastFrame * _movementspeed);
```

6. 鼠标控制同理：

```
_Mouse->capture();
float rotX = _Mouse->getMouseState().X.rel * evt.timeSinceLastFrame* -1;
```

```
float rotY = _Mouse->getMouseState().Y.rel * evt.timeSinceLastFrame * -1;
_Cam->yaw(Ogre::Radian(rotX));
_Cam->pitch(Ogre::Radian(rotY));
```

7. 最后一件事就是改变 FrameListener 实例化的部分：

```
_listener = new MyFrameListener(window， camera);
```

8. 编译运行程序。场景虽然没有改变，但是现在我们可以控制摄像机了：

刚刚发生了什么 ？

我们使用之前章节的知识添加了一个用户可以控制的摄像机。下一步要做的就是添加合成器和别的特性，使我们的程序看起来更为有趣，并学习和补充我们之前学过的知识。 】

9.11 添加合成器

之前，我们创建了三个合成器，现在我们将会给程序添加可以使用键盘响应开关闭各个合成器.

实践时刻 —— 添加合成器

我们已经几乎完成了我们的程序，我们将会添加一些合成器是我们的程序更为有趣一些：

1. 我们将会在帧监听中使用合成器，所以我们需要一个包含视口的成员变量：

```
Ogre::Viewport* _viewport;
```

2. 我们同样添加三个布尔类型的成员变量来控制合成器的开关：

```
bool _comp1， _comp2， _comp3;
```

3. 我们使用键盘输入来切换合成器的开关。为了能够区分按键，我们需要记录之前按键的状态：

```
bool _down1， _down2， _down3;
```

4. 改变 FrameListener 的构造函数来并添加视口参数：

```
MyFrameListener(Ogre::RenderWindow* win，
                                Ogre::Camera* cam，Ogre::Viewport* viewport)
```

5. 把构造函数的视口参数赋值给成员变量，并初始化所有的布尔类型的变量为 false：

```
_viewport = viewport;
_comp1 = false;
_comp2 = false;
_comp3 = false;
_down1 = false;
_down2 = false;
_down3 = false;
```

6. 如果按下 1 键，并且 1 键之前没有按过，改变其状态为按下，改变合成器的状态，并使用改变的值来激活或注销合成器，这些代码应该写在 frameStarted 函数中：

```
if(_Keyboard->isKeyDown(OIS::KC_1) && ! _down1)
{
   _down1 = true;
   _comp1 = !comp1;
   Ogre::CompositorManager::getSingleton().
```

```
                    setCompositorEnabled(_viewport， "Compositor2"，_comp1);
}
```

7. 用同样的办法来使用另外两个合成器：

```
if(_Keyboard->isKeyDown(OIS::KC_2) && ! _down2)
{
   _down2 = true;
   _comp2 = !comp2;
   Ogre::CompositorManager::getSingleton().
                setCompositorEnabled(_viewport， "Compositor3"， _comp2);
}
if(_Keyboard->isKeyDown(OIS::KC_3) && ! _down3)
{
   _down3 = true;
   _comp3 = !comp3;
   Ogre::CompositorManager::getSingleton().
                    setCompositorEnabled(_viewport， "Compositor7"， _comp3);
}
```

8. 如果刚按下的键不再按下了，我们需要改变键的状态：

```
if(!_Keyboard->isKeyDown(OIS::KC_1))
{
   _down1 = false;
}
if(!_Keyboard->isKeyDown(OIS::KC_2))
{
   _down2 = false;
}
if(!_Keyboard->isKeyDown(OIS::KC_3))
{
   _down3 = false;
}
```

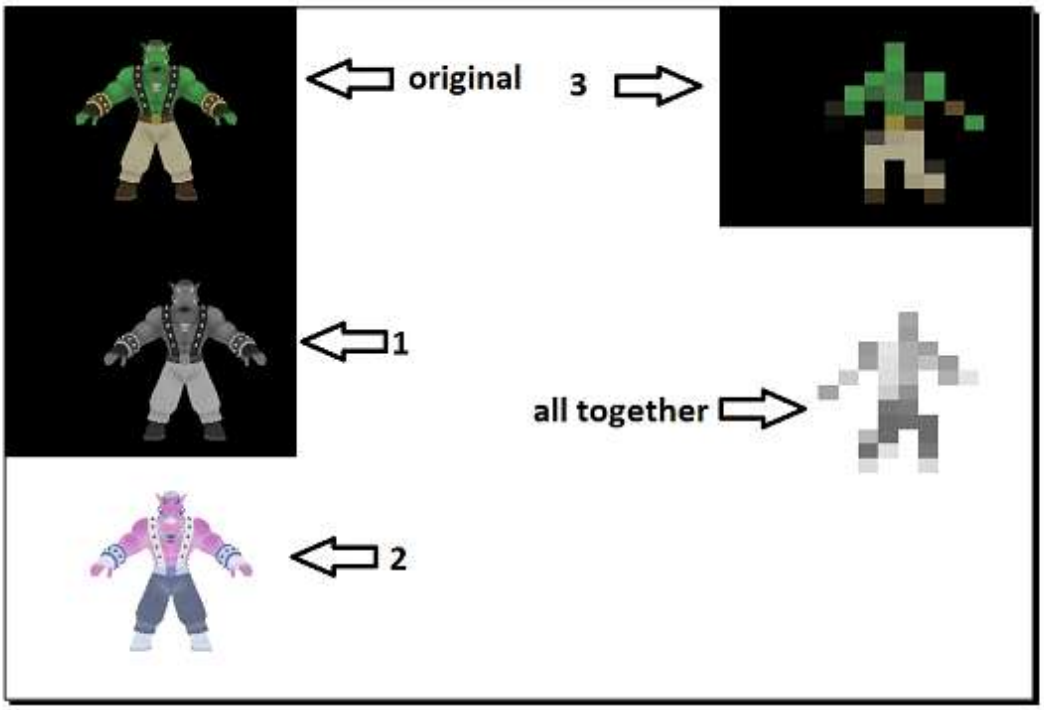
9. 在 startup()函数中，在函数的末尾添加三个合成器到视口：

```
Ogre::CompositorManager::getSingleton().
                        addCompositor(viewport，"Compositor2");
Ogre::CompositorManager::getSingleton().
                        addCompositor(viewport，"Compositor3");
Ogre::CompositorManager::getSingleton().
                        addCompositor(viewport，"Compositor7");
```

10. 记得改变 FrameListener 的实例并添加视口的指针作为参数：

```
_listener = new MyFrameListener(window，camera，viewport);
```

11 . 编译运行程序。使用 1，2，3 键，你就能自由开关不同的合成器。1 键是让图像看起来为黑白，2 键是反选图片效果，3 键是使图像看起来为更小的分辨率；你可以把任何想要的效果合成起来：



刚刚发生了什么？

我们在这章中添加一些合成器，并实现了可以使用 1，2，3 按键来开关他们。我们使用了当不只有一个合成器时，Ogre 3D 会自动合成合成器的特性。

9.12 添加平面

没有地面作为参照物，在 3D 的空间中找到方向是比较困难的，所以现在让我们添加一个地面。

实践时刻 —— 添加一个平面和一个灯光

这次所有的添加的内容是在 createScene()函数中进行的：

1. 首先我们需要添加一个平面的定义：

```
Ogre::Plane plane(Ogre::Vector3::UNIT_Y， -5);
Ogre::MeshManager::getSingleton().createPlane("plane"，
    Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME， plane，
    1500，1500，200，200，true，1，5，5，Ogre::Vector3::UNIT_Z);
```

2. 然后创建平面的实例，并绑定其到场景，并且改变其材质：

```
Ogre::Entity* ground= _sceneManager->createEntity("LightPlaneEntity"， "plane");
_sceneManager->getRootSceneNode()->createChildSceneNode()->attachObject(ground);
ground->setMaterialName("Examples/BeachStones");
```

3. 同样我们也想在场景中添加一些灯光；所以这里我们使用一个方向光：

```
Ogre::Light* light = _sceneManager->createLight("Light1");
light->setType(Ogre::Light::LT_DIRECTIONAL);
light->setDirection(Ogre::Vector3(1，-1，0));
```

4. 添加一些阴影会有更好的效果：

```
_sceneManager->setShadowTechnique(Ogre::SHADOWTYPE_STENCIL_ADDITIVE);
```

5. 编译运行程序。你将会看到一个有着石头纹理的平面，在地面上有投射有 Sinbad 的阴影。

刚刚发生了什么？

我们再一次使用了之前的知识来创建了一个平面和一个灯光，并给场景添加了阴影。

9.13 添加用户控制

我们已经在平面上有了模型实例，但是我们目前并不能移动它；让我们现在改变它吧。

实践时刻 —— 使用方向键来控制模型

现在我们通过添加用户控制模型的功能来添加场景的交互性。

1. FrameListener 需要添加两个成员变量：一个是我们想要移动的结点，另一个是我们想要移动的速度：

```
float _WalkingSpeed;
Ogre::SceneNode* _node;
```

2. 在构造函数中添加结点参数，以传递结点到构造函数中：

```
MyFrameListener(Ogre::RenderWindow* win，Ogre::Camera*
                                cam，Ogre::Viewport* viewport，Ogre::SceneNode* node)
```

3. 把结点指针赋值给成员变量并设置移动速度为 50：

```
_WalkingSpeed = 50.0f;
_node = node;
```

4. 在 frameStarted 函数中我们需要两个新的变量，这两个变量将会保存用户应用于结点的旋转和平移：

```
Ogre::Vector3 SinbadTranslate(0，0，0);
float _rotation = 0.0f;
```

5. 然后我们需要添加代码实现根据用户输入来计算位移和旋转：

```
if(_Keyboard->isKeyDown(OIS::KC_UP))
{
   SinbadTranslate += Ogre::Vector3(0，0，-1);
   _rotation = 3.14f;
}
if(_Keyboard->isKeyDown(OIS::KC_DOWN))
{
   SinbadTranslate += Ogre::Vector3(0，0，1);
   _rotation = 0.0f;
}
if(_Keyboard->isKeyDown(OIS::KC_LEFT))
{
   SinbadTranslate += Ogre::Vector3(-1，0，0);
   _rotation = -1.57f;
}
if(_Keyboard->isKeyDown(OIS::KC_RIGHT))
{
   SinbadTranslate += Ogre::Vector3(1，0，0);
   _rotation = 1.57f;
}
```

6. 然后我们需要应用平移和旋转到结点：

```
_node->translate(SinbadTranslate * evt.timeSinceLastFrame * _
WalkingSpeed);
_node->resetOrientation();
_node->yaw(Ogre::Radian(_rotation));
```

7. 程序本身需要来存储我们想要控制实体的结点指针：

```
Ogre::SceneNode* _SinbadNode;
```

8. FrameListener 实例需要这个指针

```
_listener = new MyFrameListener(window，camera，viewport，_SinbadNode);
```

9. createScene 函数需要使用上面的指针来创建和存储我们想要移动实体的结点；修改函数中的相应代码：

```
_SinbadNode = _sceneManager->getRootSceneNode()-
>createChildSceneNode();
_SinbadNode->attachObject(sinbadEnt);
```

10. 编译运行程序。你将会可以使用方向键来移动实体：

刚刚发生了什么？

我们通过在 FrameListener 添加代码，使实体可以通过方向键来移动。现在我们的 Sinbad 可以像魔术师一样来平面中飘动了。 】

9.14 添加动画

让模型无动作的平移不是我们所想要的；所以让我们添加一些动画吧。

实践时刻 —— 添加动画

我们的模型可以移动，但是现在却不能做动作，让我们接下来改变它：

1. FrameListener 类需要两个动画状态：

```
Ogre::AnimationState* _aniState;
Ogre::AnimationState* _aniStateTop;
```

2. 为在构造函数中获得动画状态，我们需要一个实体的指针：

```
MyFrameListener(Ogre::RenderWindow* win，Ogre::Camera*
cam，Ogre::Viewport* viewport，Ogre::SceneNode* node，Ogre::Entity* ent)
```

3. 通过使用这个指针我们可以检索动画状态，并在接下来的使用中保存他们：

```
_aniState = ent->getAnimationState("RunBase");
_aniState->setLoop(false);
_aniStateTop = ent->getAnimationState( «RunTop »);
_aniStateTop->setLoop(false);
```

4. 既然我们有了 AnimationState，我们需要在 frameStarted 函数中有一个开关，这个开关描述了实体这帧是否移动了。我们添加这个开关到 if 判断中来查询键盘的状态：

```
bool walked = false;
if(_Keyboard->isKeyDown(OIS::KC_UP))
{
   SinbadTranslate += Ogre::Vector3(0，0，-1);
```

```
    _rotation = 3.14f;
    walked = true;
}
if(_Keyboard->isKeyDown(OIS::KC_DOWN))
{
    SinbadTranslate += Ogre::Vector3(0，0，1);
    _rotation = 0.0f;
    walked = true;
}
if(_Keyboard->isKeyDown(OIS::KC_LEFT))
{
    SinbadTranslate += Ogre::Vector3(-1，0，0);
    _rotation = -1.57f;
    walked = true;
}
if(_Keyboard->isKeyDown(OIS::KC_RIGHT))
{
    SinbadTranslate += Ogre::Vector3(1，0，0);
    _rotation = 1.57f;
    walked = true;
}
```

5. 如果模型移动了，我们激活动画；如果动画结束，我们重新使它循环

```
if(walked)
{
    _aniState->setEnabled(true);
    _aniStateTop->setEnabled(true);
    if(_aniState->hasEnded())
    {
        _aniState->setTimePosition(0.0f);
    }
    if(_aniStateTop->hasEnded())
```

```
    {
        _aniStateTop->setTimePosition(0.0f);
    }
}
```

6. 如果模型没有移动，我们注销动画，并设置动画为其开始的位置：

```
else
{
    _aniState->setTimePosition(0.0f);
    _aniState->setEnabled(false);
    _aniStateTop->setTimePosition(0.0f);
    _aniStateTop->setEnabled(false);
}
```

7. 在每一帧中，我们需要添加经过的时间给动画；否则，它就不会移动：

```
_aniState->addTime(evt.timeSinceLastFrame);
_aniStateTop->addTime(evt.timeSinceLastFrame);
```

8. 程序需要一个指向实体的指针

```
Ogre::Entity* _SinbadEnt;
```

9. 我们在初始化 FrameListener 的时候使用了这个指针：

```
_listener = new MyFrameListener(window，camera，viewport，_SinbadNode，_SinbadEnt);
```

10. 同样，创建实体：

```
_SinbadEnt = _sceneManager->createEntity("Sinbad.mesh");
```

11. 编译运行程序。当模型移动的时候就可以播放相应的移动动画了：

刚刚发生了什么？

我们给我们的模型添加了动画并设置在模型运动的时候运行。

动手试试 —— 查阅我们使用了什么

查阅我们上一个例子所用到技术的相关章节。

9.15 总结

本章中我们学习了很多关于激活我们程序和运行我们 Ogre 3D 程序的相关知识。

特别的，我们包含了以下几个方面：

1. Ogre 3D 是怎么运行的。
2. 怎样使我们的渲染主循环运行。
3. 写出我们自己的一个应用和帧监听的实现

一些专题我们前面已经涉及到了，但是现在我们把它们结合在一起形成一个更加复杂的应用。

现在我们已经学习了创建我们自己的 Ogre 3D 应用所需要的所有内容。下一章节中我们会把重点放在使用其他的库或者一些附加的功能使应用程序更加完美。

第十章 粒子系统和 Ogre 3D 扩展

这是这本书的最后一章，我们将会接触到我们未曾涉及的一章——粒子系统。在此之后，这章将会呈现一些Ogre 3D的扩展和特效手法，这在未来对于你来说会是很重要的，但是不是对所有的程序都必要。

在这一章，我们将会：

1. 学习什么是粒子系统和怎样使用它
2. 创建几个不同的粒子系统
3. 了解Ogre 3D扩展
4. 自豪的读完这本书

那么让我们开始吧

10.1 增加一个粒子系统

我们将绑定一个烟雾特效粒子系统到 sinbad 上，这有助于我们观察它所在的位置.

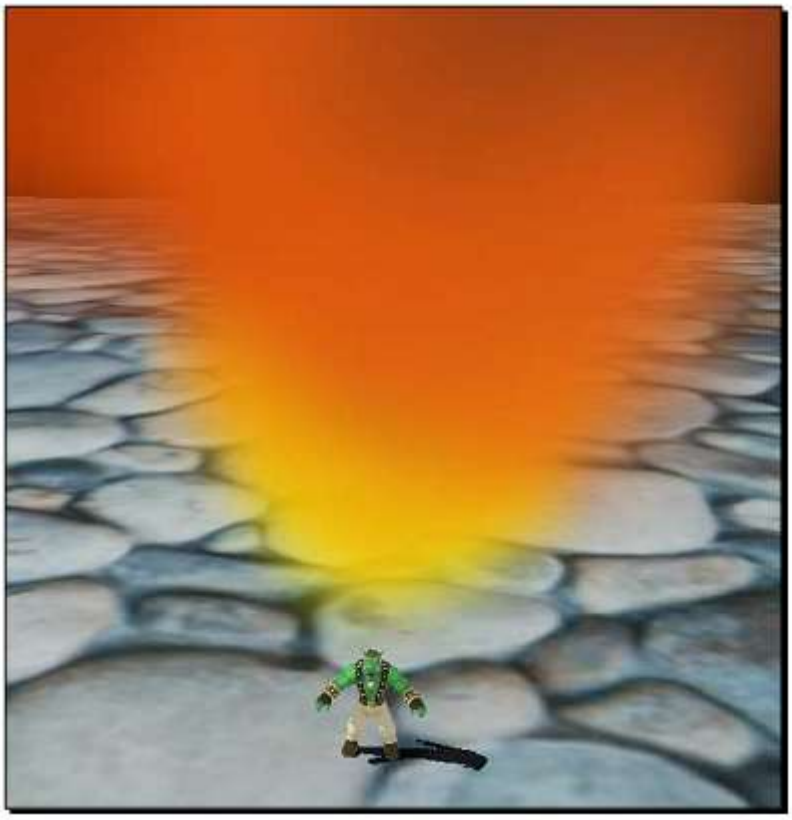
实践时刻 —— 添加烟雾

我们将使用之前最后一个例子的代码：

1. 创建一个预先定义粒子脚本的粒子系统。增加这个粒子系统绑定到和 sinbad 实体的相同结点上：

```
Ogre::ParticleSystem* partSystem =
                    _sceneManager->createParticleSy stem("Smoke"， "Examples/Smoke");
_SinbadNode->attachObject(partSystem);
```

2. 编译运行程序。将会看到Sinbad产生大量的烟雾。



刚刚发生了什么？

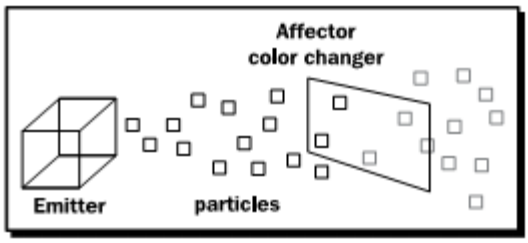
我们已经定义过例子脚本来创建一个粒子系统。这样，这样粒子系统跟随着我们的实体一起运动。

什么是粒子系统?

在我们创建自己的粒子系统而不是载入预先定义好的，我们需要讨论一下到底什么是粒子系统。我们已经看见了粒子系统创建带来的效果——在我们的实例中，是一个烟雾锥形体；但是它是怎么被创建的？

一个例子系统由两到三个不同的部分组成——发射器，粒子和效应器（可选择的）。这三部分最重要的是粒子本身，正如它的名字粒子系统一样。一个粒子通过一个四边形显示一个颜色或者纹理，或者显卡的点渲染能力。当粒子使用一个四边形，这个四边形总是经旋转后正对着摄像机。每个粒子拥有一组参数，包括生存时间，方向和速率。虽然还有很多其它的参数，但是这三个是粒子系统概念里最重要的三个参数。参数生存时间控制着一个粒子的生存和死亡。一般，一个粒子在它被销毁前是存活不了几秒种的。这个效果可以在我们之前的烟雾例子中看到。将会有一个粒子消失点。对于这些粒子，生存计时器得到0并且这些粒子被销毁。

方向和速率描述粒子的移动行为。在这个烟雾例子中，我们的方向是向上。发射器创建一个预先定义每秒钟产生的粒子数量，可以看做是粒子的来源。效应器，从另一方面来说，不创建粒子但改变它们的一些参数。在这个场景中我们并没有看到任何的效应器，但之后将会应用到。一个效应器能够改变粒子方向，速率，或者发射器创建的粒子颜色。



现在我们已经知道了一些基础概念，那么让我们自己手动创建一些粒子系统吧。

10.2 创建一个简单的粒子系统

创建一个粒子系统，我们需要定义整个系统的行为，尤其是发射器行为。

实践时刻 ——创建一个简单的粒子系统

创建一个粒子系统，我们需要定义整个系统的行为，尤其是发射器行为。

1. 粒子系统在.particle文件中定义。在mendia/particle文档中创建一个。

2. 定义这个系统并命名为MySmoke1:

```
particle_system MySmoke1
{
```

3. 每个粒子应当使用Example/Smoke材质并且10个单位长10个单位宽：

```
material              Examples/Smoke
particle_width        10
particle_height       10
```

4. 我们想要一个最大值为500粒子同时每个粒子都应总是正对着摄像机的点：

```
quota                500
billboard_type       point
```

5. 我们想要一个发射器，它从一个单独点上以每秒3个粒子的速率发射粒子

```
emitter Point
   {
emission_rate 3
```

6. 粒子应当在（1，0，0）方向上以20单元每秒的速度发射：

```
direction 1 0 0
velocity 20
```

7. 脚本代码就这么多。闭合括号

```
   }
}
```

8. 在 *createScene()* 函数中，把

```
Ogre::ParticleSystem* partSystem =
                _sceneManager->createParticleSystem("Smoke"，"Examples/Smoke");
```

修改为：

```
Ogre::ParticleSystem* partSystem =
                _sceneManager->createParticleSystem("Smoke"，"MySmoke1");
```

9. 编译和运行。你应当看到一个 sinbad 和尾随着一条烟雾尾巴。

刚刚发生了什么？

我们创建了第一个粒子系统。对于它，我们需要一个.particle文件来存储脚本代码。在这个脚本代码中，我们以关键词particle_system开始来定义粒子系统，然后是我们想要的命名名称，就像其它脚本那样。在第3步中，我们定义了粒子材质。我们使用了SDK自带的材质。这个材质仅绑定一个纹理并结合这个纹理到顶点颜色上而忽略其它灯光。下面是完整的材质脚本：

```
material Examples/Smoke
{
   technique
   {
      pass
      {
         lighting off
         scene_blend alpha_blend
         depth_write off
         diffuse vertexcolor
         texture_unit
         {
            texture smoke.png
            tex_address_mode clamp
         }
      }
   }
}
```

我们给予每个粒子10单元的长度和宽度。第4步，定义了我们想要任意给定的时刻生存在粒子系统中的粒子最大数量；这个数量有助于阻止定义一个错误的粒子系统而导致我们整个程序的运行缓慢。如果达到了这个数量，发射器将不被允许创建新的粒子。这一步也定义了我们想要作为粒子的点总是面向摄像机。第5步增加了一个从确定的点中每秒发射3个粒子的发射器。第6步设置了粒子移动的方向和速率。然后我们改变程序使用新的粒子系统，最后观察它。

10.3 一些更多的参数

现在我们有个一个可测试的粒子系统，让我们尝试一些别的参数。

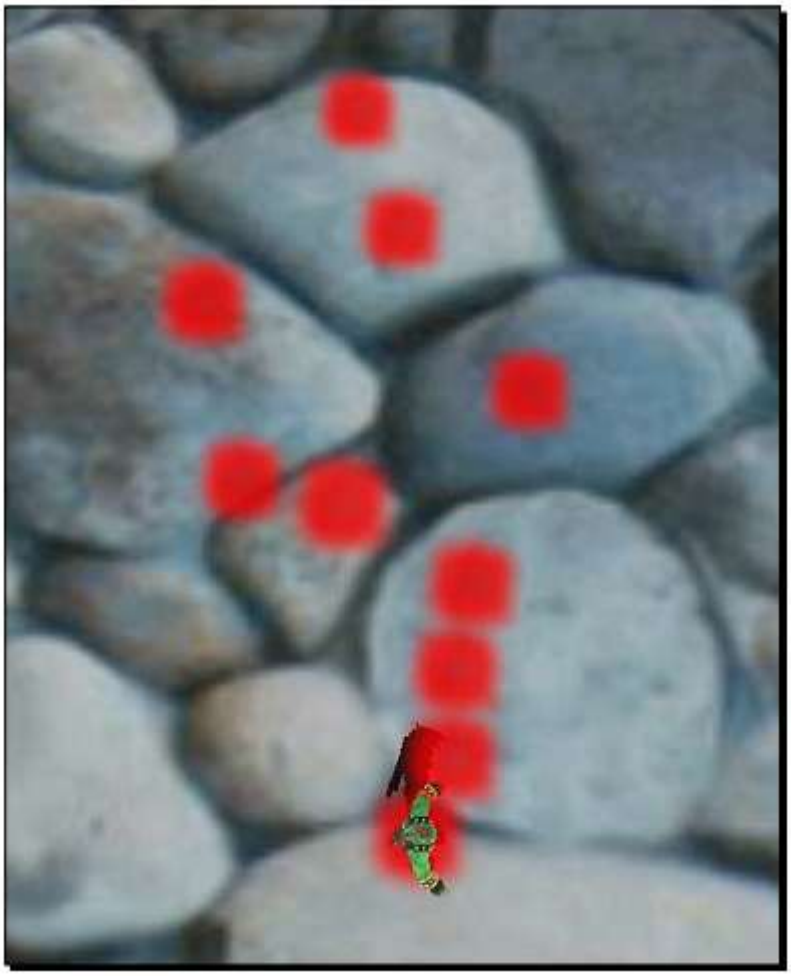
实践时刻 ——我们将增加一些新的参数。

我们将增加一些新的参数。

1. 用下面三个新参数来增加一个点发射器：

```
angle 30
time_to_live 10
color 1 0 0 1
```

2. 编译运行。你应当看到不同方向上飞行的红色粒子



刚刚发生了什么？

我们增加了三个改变粒子系统行为的参数。现在粒子是红色的在不同方向上飞。参数angle定义了每创建一个粒子需要多少角度才能够从给定的方向上区别开来。Ogre 3D使用了一个随机生成器生成在一定范围内的方向。因为这个方向可以最大到30度，其中的一些粒子能够飞进地面。

参数time_to_live设置了每个粒子的生命周期，在我们的例子中，为10s。默认值为5s。通过这个，我们增加了每个粒子一倍的生命周期，所以我们可以更长久地观察他们的行为了。

参数color设置了粒子的顶点颜色为给定的颜色向量，在我们的例子中，为红色。

简单测试 —— 什么可以创建一个粒子系统

说出三个创建粒子系统的组件;哪个是可选的?

10.4 别的参数

我们可以创建多种不同的粒子系统。那么让我们创建多一些吧。

实践时刻 —— 生存时间和颜色范围

同样我们将增加一些其它参数来观察它们的效果。

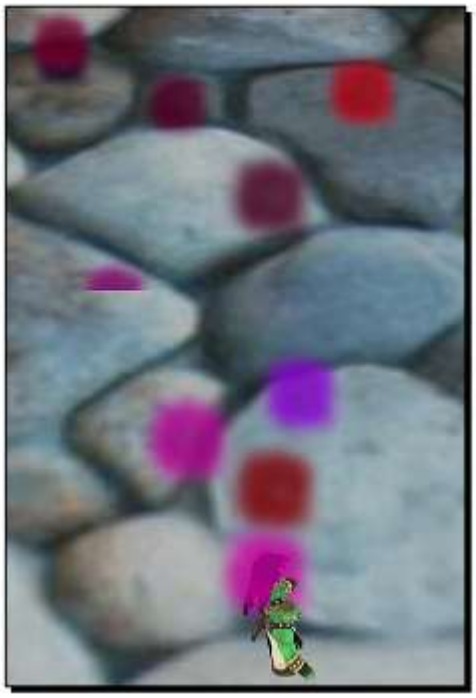
1. 改变time_to_live为一个最大和最小值范围：

```
time_to_live_min 1
time_to_live_max 10
```

2. 改变color的值用相同的方法

```
color_range_start 1 0 0
color_range_end 0 0 1
```

3. 调整程序代码；然后编译运行。你将看到不同颜色粒子和一些将消失在另一些前面的粒子。



刚刚发生了什么？

我们使用了描述一个范围值的参数来代替使用单独的参数，然后让ogre 3D选择确切的值。这增加了我们的粒子系统的多样性并可用于模型更实际的自然效果，因为实际上，很少有东西在时间和空间上有一个相同的外观。

简单测试 —— 生存时间

用自己的话描述time_to_live和time_to_live_min之间的区别

10.5 新的开关

尝试使用更多的参数.

实践时刻 —— 添加粒子系统的间隔时间

现在我们将看到存在的一些参数它不影响粒子的外观，而只影响它们的发射方式。

1. 移除在point emitter里增加的参数，只保留 `emission_rate`， `direction`， 和 `velocity`:

```
emitter Point
{
    emission_rate 30
    direction 1 0 0
    velocity 20
```

2. 然后增加发射一个粒子所需时间的参数和重新开始之前需要等待多久的参数。

```
    duration 1
    repeat_delay 1
}
```

3. 编译运行。你应当看到一股白色颗粒，它产生的效果是通过每次打断一下发射器发射而产生的。

刚刚发生了什么？

我们增加了参数duration，它定义了在发射结束之前发射器发射粒子多长时间。Repeat_delay设置了发射器重新发射粒子之前需要等待多长时间。通过这两参数，我们已经定义了一个发射1s等待1s后开始发射的发射器。

简单测试 —— 发射参数

尝试写出所有12个发射器参数和它们是怎样影响发射器的。

10.6 添加效应器(affector)

我们已经当我们创建和使用发射器的时候改变粒子的行为和外观。现在我们将使用效应器，它将在整个粒子生存周期里改变外观和行为。

实践时刻 —— 添加一个展示效应器

1. 为了展示效应器，我们需要一个简单的point发射器，它每秒以20单元速度发射30个粒子并且生存周期为100s：

```
emitter Point
{
   emission_rate 30
   direction 1 0 0
   velocity 20
   time_to_live 100
}
```

2. 在粒子的整个生存周期里，我们想要它每秒增长五倍的大小。为了做到这点，我们增加一个Scaler效应器：

```
affector Scaler
{
   rate 5
}
```

3. 重新编译运行。你应看到生存时间中每秒变大的粒子。

刚刚发生了什么？

我们增加了一个效应器来改变粒子的大小。Scaler效应器通过给定的值来缩放粒子。在我们的例子中，每个粒子的大小通过一个每秒5倍的因数来缩放。

10.7 改变颜色

我们已经改变了大小，现在来改变下粒子的颜色。

实践时刻 —— 改变颜色

1. 替换scaler，增加一个每一个颜色通道每秒减去0.25的ColorFader效应器

```
color channel per second:
affector ColorFader
{
   red -0.25
   green -0.25
   blue -0.25
}
```

2. 编译运行。你应当看到粒子在它的生存周期里怎样慢慢的从白变黑。

刚刚发生了什么 ？

我们增加了一个改变每种颜色通道的效应器，使用了一个预先定义的值。

动手试试 —— 改变颜色为红色

改变colorfader代码，让粒子从白褪到红色。结果应该看起来如下图

10.8 双向改变

我们已经改变一种颜色到另一种，但是有时我们想要这个改变依赖于粒子的生存周期。当构建火和烟雾模型时候这是很重要的。

实践时刻 ——依赖于生存时间的粒子的颜色改变

我们现在将通过粒子效应器来介绍更多的颜色。

1. 对于这个例子，我们不想我们的粒子存活100s，所以改变生存周期为4s

```
emitter Point
{
   emission_rate 30
   direction 1 0 0
   velocity 20
   time_to_live 4
}
```

2. 因为我们想要一个稍微不同的行为，所以我们将使用第二个可用的colorfader。它应当每秒每个颜色通道褪色一个单元的颜色：

```
affector ColorFader2
      {
            red1 -1
            green1 -1
            blue1 -1
```

3. 现在，当粒子只有2s存活时间，代替减法的颜色通道，增加相同的值“

```
   state_change 2
   red2 +1
   green2 +1
   blue2 +1
}
```

4. 重新编译运行程序

刚刚发生了什么 ？

我们使用了colorfader2效应器。这个效应器首先通过给予red1，green1和blue1的值来改变，当粒子只有state_change参数的生存时间的时候。如red2，green2，blue2的这些值被修改直到粒子死亡。在这个例子中，我们使用了这个效应器来先改变粒子的颜

色从白到黑，然后当距离死亡2s时，我们改变它从黑到白。

10.9 更多复杂的颜色操作

一种更加复杂的创建粒子颜色的操作方法。

实践时刻 —— 使用更加复杂的颜色操作

再一次，我们使用粒子颜色并影响它们。

1. 我们将使用一个新的效应器命名为 `ColorInterpolator`：

```
affector ColorInterpolator
{
```

2. 当它产生的时候我们确定像素应当使用哪种颜色。我们将使用白色：

```
color0 1 1 1
```

3. 当粒子存活到它生存周期的四分之一的时候，它应当为红色：

```
time1 0.25
color1 1 0 0
```

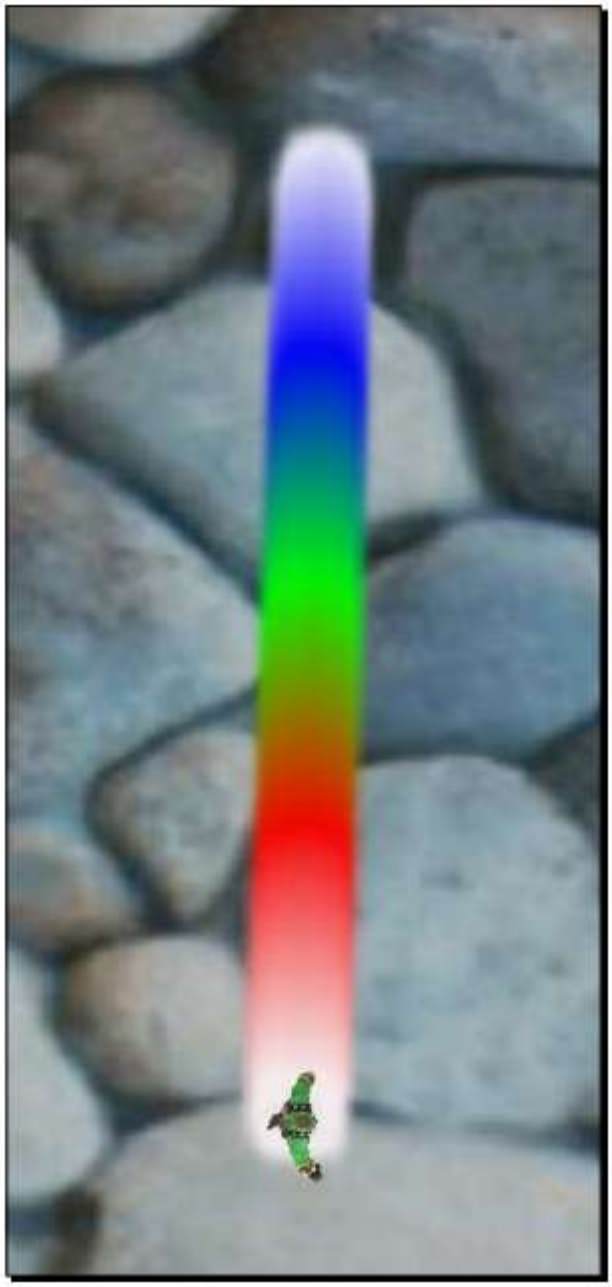
4. 在四分之一周期到二分之一生存周期里，我们想要它为绿色：

```
time2 0.5
color2 0 1 0
```

5. 在二分之一到四分之三周期，它应当为蓝色，最后它又变回白色：

```
time3 0.75
color3 0 0 1
time4 1
color4 1 1 1
```

6. 编译运行。你应当看到粒子流，它们应当从白到红到绿到蓝最后变回白色。



刚刚发生了什么 ？

我们使用了另一个效应器来创建一个更复杂的颜色操作。ColorInterpolator操控这所有粒子颜色。我们通过关键词timeX和colorX来定义这个操作，X必须是0到5之间。time0的0表示我们在粒子被创建的时候想要效应器的颜色color0。time1的0.25意味着我们想要效应器使用color1当粒子存活到它生命周期的四分之一。在这两个时间点之间，效应器应会插值来达到渐变的效果。我们的例子定义五个点，并且每个点有一个不同的颜色。第一个和最后一个使用白色，第二个点使用红色，第三个使用绿色，第四个使用蓝色。每个点都是生存周期的四分之一，所以在整个生存周期，每种颜色都使用了相同的时间段。

10.10 增加随机性

为了创建一个更漂亮的效果，可以通过增加一点随机性给我们的粒子系统，这样它就会看起来比较自然。

实践时刻 —— 添加随机性

增加随机性能够提到一个场景的视觉效果，所以让我们来实现它。

1. 移除 `ColorInterpolator` 效应器。

2. 增加一个不同的效应器命名为 `DirectionRandomiser`:

```
affector DirectionRandomiser
{
```

3. 首先我们定义效应器在每个粒子的轴上的影响值：

```
randomness 100
```

4. 然后我们描述每次效应器被应用的时候，有多少粒子应当被影响。1.0代表100%，0代表0%。然后我们确定一下是否我们想要我们的粒子保持它们速率或者是否速度也应当被改变：

```
    scope 1
    keep_velocity true
}
```

5. 编译运行。这次，我们不应当看到一个单独的粒子流，而是，很多的粒子飞行在不确定的方向上，但是大概在一个方向上。

刚刚发生了什么 ？

DirectionRandomiser效应器通过给粒子的不同值，从而改变了粒子的方向。通过这个效应器，它能够给粒子的运动增加一个随机组件。

10.11 偏移

我们将尝试的最后一个效应器是使粒子在它们的方向上模拟一个障碍的平面而对其造成的偏移效果。

实践时刻 —— 使用偏移平面

为了能够让我们的粒子能在平面反弹起作用，这将是我们在这里将要做的。

1. 代替随机性发射器，我们使用DeflectorPlane效应器

```
affector DeflectorPlane
{
```

2. 这个平面定义使用了一个在空间中的点和这个平面的法线：

```
plane_point 0 20 0
plane_normal 0 -1 0
```

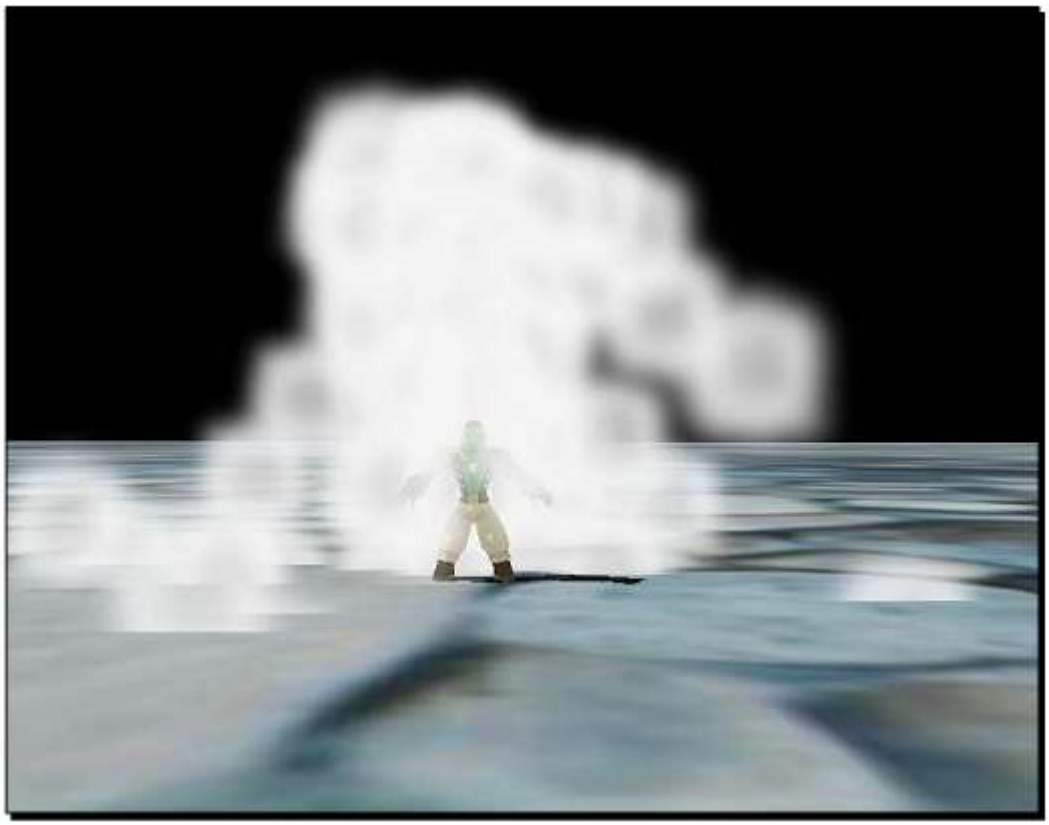
3. 最后定义当粒子撞击这个平面时，粒子应当受到什么影响。我们想要它们保持它们原先的速度，所以我们选择1.0作为值：

```
    bounce 1.0
}
```

4. 为了观察偏移平面的效果，我们需要我们的粒子朝稍微不同的方向上移动。所以更改发射器，使粒子的发射方向朝30度以内的方向发射。此外，由于平面悬浮于空中，我们的粒子应使用向上的方向向量来初始化粒子方向。

```
emitter Point
{
    emission_rate 30
    direction 0 1 0
    velocity 20
    time_to_live 4
    angle 30
}
```

5. 编译运行。粒子在空中的隐形平面反弹。

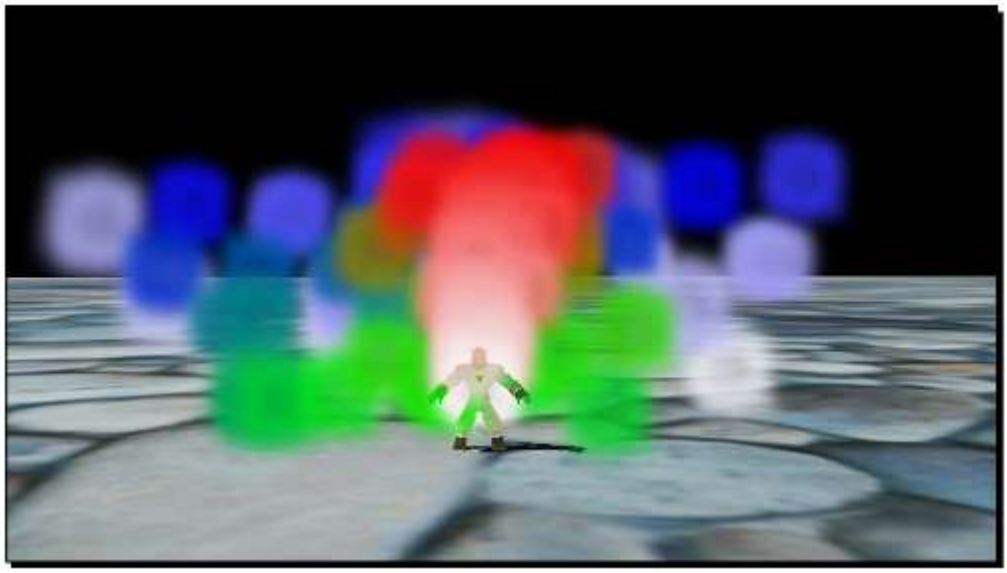


刚刚发生了什么 ？

我们增加了一个悬浮于空间的平面，并同时保持粒子的速度。

动手试试 —— 让我们做更多

创建一个新的程序，让第二个平面位于（0，0，0）使得它们的粒子受到第一个平面的影响。同时增加ColorInterpolator效应器。效果应当看起来和下图一样：



10.12 其他类型的发射器

我们已经使用了一个点发射器，但是，当然，还有各种不同的发射器类型供我们使用。

实践时刻 —— 使用一个盒发射器

从一个点上发射粒子是很单调的，使用一个box来发射会更有趣。】

1. 改变发射器类型从发射点变为发射盒

```
emitter Box
{
```

2. 定义一个用于创建粒子的 box：

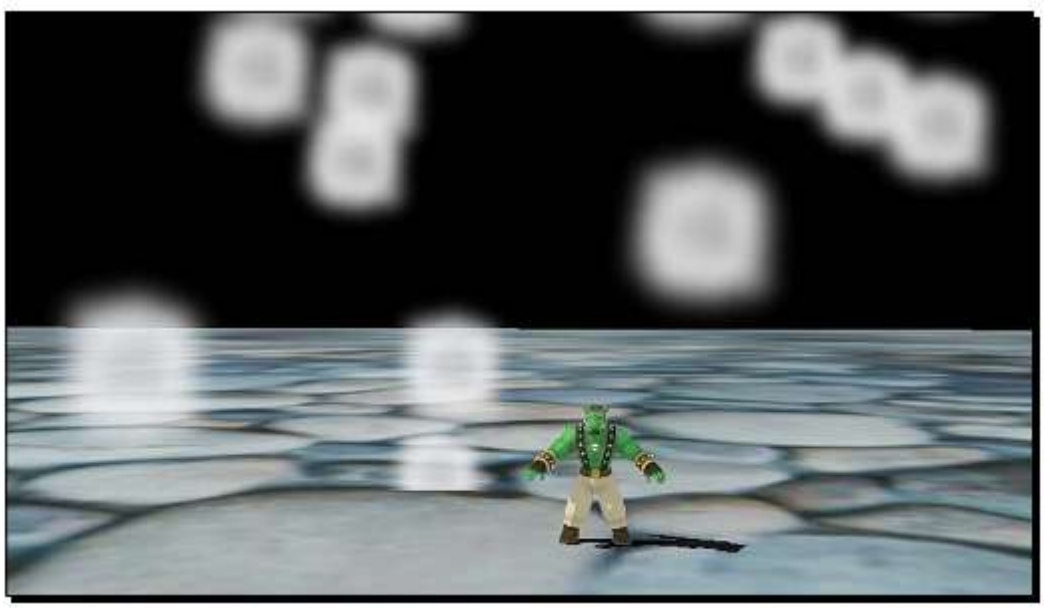
```
height 50
width 50
depth 50
```

3. 让发射器创建每秒10粒子并且它们的移动速度为20:

```
emission_rate 10
direction 0 1 0
```

```
    velocity 20
}
```

4. 使用新的粒子系统，编译运行。你应当看到粒子被创建在sinbad的周围，飞向空中。



刚刚发生了什么 ？

我们使用了另一个发射器类型，在这个例子中，盒发射器(Box emitter)。我们定义了一个box，并且这个发射器使用了随机点在它的box内部作为起始位置来创建粒子。这个发射器能够用于创建不从准确点发射的粒子系统，而是从一个范围上发射。如果我们需要一个用于粒子发射的平面或一条线，我们只需要相应地设置box的参数即可。

10.13 使用环形发射

除了box类型，还有其它发射器类型，如环。

实践时刻 —— 使用环来发射粒子

代替一个点或者盒，我们可以使用一个环作为发射器。

1. 改变发射器类型为Ring

```
emitter Ring
```

```
{
```

2. 通过width和height来定义Ring的长宽

```
height 50
width 50
```

3. 现在，为了创建一个环而不是一个圈，我们需要定义其内部部分不发射粒子。这儿我们使用百分比：

```
inner_height 0.9
inner_width 0.9
```

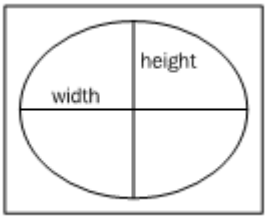
4. 余下部分没有改动，如下：

```
    emission_rate 50
    direction 0 1 0
    velocity 20
}
```

5. 编译运行。移动摄像机到模型的顶部，你应当看到发射粒子的环。

刚刚发生了什么 ？

我们在一个定义好的环形内使用了环发射器发射粒子。为了定义这个环形，我们使用了长宽，而不是一个点和半径。长宽描述了椭圆长宽的最大值。这里，下图展示了环是怎样定义的。通过inner_width和inner_height，我们定义环形内部多少是不发射粒子的。这里我们不使用空间单位，而是百分比。



10.14 在最后，我们想要一些烟火

这将是本书的最后一个粒子，所以放点烟火是较好的表示。

实践时刻 —— 添加一些烟火

在一件特别的事件后来场烟花是非常赞的。

1. 创建一个在任意方向上以稳定时间间隔爆炸不同颜色的粒子的粒子系统：

```
particle_system Firework
{
   material                Examples/Smoke
   particle_width       10
   particle_height     10
   quota                    5000
   billboard_type       point
   emitter Point
   {
   emission_rate 100
      direction 0 1 0
      velocity 50
      angle 360
      duration 0.1
      repeat_delay 1
      color_range_start 0 0 0
      color_range_end 1 1 1
   }
}
```

2. 创建5个这种粒子系统实例：

```
Ogre::ParticleSystem* partSystem1 =
            _sceneManager->createParticleSystem("Firework1"，"Firework");
Ogre::ParticleSystem* partSystem2 =
            _sceneManager->createParticleSystem("Firework2"，"Firework");
Ogre::ParticleSystem* partSystem3 =
            _sceneManager->createParticleSystem("Firework3"，"Firework");
Ogre::ParticleSystem* partSystem4 =
            _sceneManager->createParticleSystem("Firework4"，"Firework");
Ogre::ParticleSystem* partSystem5 =
            _sceneManager->createParticleSystem("Firework5"，"Firework");
```

3. 然后创建五个在空中不同位置的结点：

```
Ogre::SceneNode* node1 = _sceneManager->getRootSceneNode()->
                                        createChildSceneNode(Ogre::Vector3(0，10，0));
Ogre::SceneNode* node2 = _sceneManager->getRootSceneNode()->
                                        createChildSceneNode(Ogre::Vector3(10，11，0));
Ogre::SceneNode* node3 = _sceneManager->getRootSceneNode()->
                                        createChildSceneNode(Ogre::Vector3(20，9，0));
Ogre::SceneNode* node4 = _sceneManager->getRootSceneNode()->
                                        createChildSceneNode(Ogre::Vector3(-10，11，0));
Ogre::SceneNode* node5 = _sceneManager->getRootSceneNode()->
                                        createChildSceneNode(Ogre::Vector3(-20，19，0));
```

4. 最后，绑定粒子系统到结点上：

```
node1->attachObject(partSystem1);
node2->attachObject(partSystem2);
node3->attachObject(partSystem3);
node4->attachObject(partSystem4);
node5->attachObject(partSystem5);
```

5. 编译运行，享受一下。

刚刚发生了什么 ？

我们创建了一个激情四射烟火的粒子系统并复制了它，让它看起来像是有好多烟火在空中。

简单测试 —— 不同类型的发射器

写出所有我们在这章使用到的发射器类型，简述一些它们的不同和共同点。

10.15 OGRE 3D 扩展程序

我们已经看到许多Ogre 3D提供的不同功能，但Ogre 3D也很易于扩充它新的功能。所以有很多不同的库，用来增加一些新的功能给Ogre 3D。我们将讨论这些库，描述下有什么附加功能。一个完整库的列表，可以在维基百科获得：www.Ogre3D.org/tikiwiki/OGRE+Libraries

Speedtree

Speedtree是一个用以渲染很多漂亮树和草的商业解决方案。它被广泛地应用几个商业游戏和Ogre 3D的创立者Sinbad提供给Ogre 3D一个版本。Speedtree的Ogre 3D版本须购买，不可以免费使用的。更多的信息，可以发现在http://www.ogre3d.org/tikiwiki/OgreSpeedtree 。

Hydrax

Hydrax增加是一个附加的功能，它可以为Ogre 3D渲染漂亮的水场景。通过这个附加功能，水能够被增加到场景上并且可以进行很多不同的设置，例如水的深度设置、增加泡沫、水下光线的影响，等等。附加功能参见：http://www.ogre3d.org/tikiwiki/Hydrax .

Caelum

Caelum是另一个附加扩展，主要介绍Ogre 3D天空昼夜循环渲染。它使太阳和月亮根据一个日期和时间来地渲染。它也使天气效果比如雪或雨，一个复杂的云仿真让天空看起来像真实越好。更多参见维基百科：http://www.ogre3d.org/tikiwiki/Caelum。

Particle Universe

另一个商业附加库是Particle Universe。Particle Universe增加一个新的粒子系统给ogre 3D，它提供比ogre 3D粒子系统更多允许的粒子特效。同时，它带有一个粒子编辑器，允许美工在一个单独的程序中创建粒子，然后程序员可以载入这个创建好的粒子脚本。这个插件参见于：http://www.ogre3d.org/tikiwiki/Particle+Universe+plugin

10.16 GUIs

有很多不同的GUI库可供Ogre 3D使用，每一种都有它存在的理由，但目前并没有一个GUI库每个人都使用。最好的办法是尝试了其中的一些，然后再做决定选择我们需要的库。

CEGUI

CEGUI可能是第一个被整合到Ogre 3D中的库。它提供了一个GUI系统应有的函数甚至更多。它提供了一个于在代码之外GUI编辑器用来创建您的图形用户界面和为你的GUI定制不同的皮肤。更多的信息参见：http://www.cegui.org.uk/wiki/index.php/Main_Page .

BetaGUI

BetaGUI是极其微小的库，它是在一个头和一个cpp的文件。唯一的依赖是Ogre 3D，并能提供基本的功能如同创建窗口，按钮，文本域，和静态文本。这不是一个完整的图形用户界面，但是它提供了基本的功能无效其它任何依赖库，因此可以用在一个简单而快速的解决方案。在http://www.ogre3d.org/tikiwiki/BetaGUI 都能找到。

QuickGUI

QuickGUI是比BetaGUI一种更复杂和强大的方案。 虽然QuickGui提供更多的窗体小部件，它也让一体化进程变得更为困难。QuickGUI是一个成熟的GUI解决方案，可用于各种不同的项目，定期更新。维基百科网站能找到http://www.ogre3d.org/tikiwiki/QuickGUI

Berkelium

Berkelium不是一个GUI库等，因为它没有任何部件或类似的情况。相反，它使 Ogre 3D渲染使用谷歌Chromium库。在这个库的帮助下，使得建立一个游戏浏览器变为有可能。更多参见：http://www.ogre3d.org/tikiwiki/Berkelium 。

10.17 总结

我们这章学习了很多东西。

特别，我们涉及了：

* 怎样通过不同类型的发射器创建粒子系统
* 效应器是怎样影响发射器的
* Ogre 3D的一些附加扩展库

附录

这是这本书的结束，我想向您表示祝贺。我知道阅读一本完整的编程的书是需要做大量工作的，并且做完所有的例子，去理解一个新话题，但它也非常有益而且新知识永远属于你。我希望你会喜欢这本书，它教给你足够的能创建自己的交互式的3D应用程序的方法，因为，在我看来，这是程序和计算机科学领域中最有趣并且高速更新的一个领域。

简单测试答案

Chapter 1

Installing Ogre 3D

1 a which post effects are shown

in the samples

Bloom， Glass， Old TV， Black and White， and Invert

2.1 b

and

c

which libraries to link OgreMain.lib and OIS.lib

2.2 which libraries to link Add _d after the library name

3 c ExampleApplication

and how to display a model

Create an entity using the createEntity()

function of the SceneManager and then attach

this entity to a scene node

Chapter 2

Setting up the Environment

1 a finding the

position of

scene nodes

MyEntity will be at (60，60，60) and MyEntity2 will be at (0，0，0)

2 b playing with

scene nodes

(10，-10，10)

3 b rotating a

scene node

pitch， yaw， roll

4 creating child

scene nodes

4.1)One way is to only give a name for the scene node and the other

one is to give a name and a position where the scene node should be

created.

4.2) Please refer the code.

5 b even more

about the

scene graph

From the root to the leafs

6 Ogre3D and

spaces

The three spaces are world， parent， and local.

Chapter 3

Felix Gogo

1

different

light

sources

A point light is like a light bulb and a spotlight is like a flashlight

2 different

light types

Point， Spot， and Directional.

Chapter 4

Felix Gogo

1

c design pattern of

FrameListener

Observer pattern

2 the difference between

time- and frame-based

movement

When using frame-based movement， the entity is

moved the same distance each frame， by time passed

movement， the entity is moved the same distance

each second.

3 window questions A window handle is a unique identifier used and

created by the operating system to manage its

windows， we need the handle of our application

window to receive the input events our window

gets in focus.

4 capturing the input To get the newest state the keyboard has

Chapter 5

The Book Inventory Bundle

1 the

importance

of time

Because this way the animation is independent from the real time that

has passed. This also enabled us to run the same animation at different

speeds.

Chapter 7

The Bookshelf: First Stab

1 texture

modes

How texture coordinates are handled that are lower or higher than the

range of 0 to 1

Chapter 9

The Ogre 3D Startup Sequence

1 the three

event

handlers

frameStarted which gets called before the frame is rendered

frameRenderingQueued which is called after the frame is rendered

but before the buffers are swapped and

frameEnded which is called after the current frame has been rendered

and displayed.

Chapter 10

How About a Graphical Interface?

1 what makes a

particle system

Particle， Emitter， and optional Affector

2 emitter

parameters

emission_rate: How many particles should be emitted

per second

direction: In which direction the particles should move

velocity: At which speed they should move

duration: How long does the emitter emit particles

repeat_delay: How long until it start emitting again

time_to_live: The length of the life of a particle

time_to_live_min: The minimum lifespan of a particle

time_to_live_max: The maximum lifespan of a particle

angle: How much the particles' movement direction can

differ from the direction given

colour: The color of a particle an particle has

colour_range_start: Beginning point for the particle's

color interpolation

colour_range_end: End point for the particle's color

interpolation

索引