文档:教程:基础教程:基础教程一

出自 Ogre3D 开放资源地带

跳转到: 导航, 搜索

目录

简介

在这篇教程里，我会向您介绍 OGRE 最基础的构架：场景管理器，场景节点和实体。由于我需要在这篇教程里把 OGRE 的基本概念介绍给你，所以我们不会接触太多的代码。在您阅读这篇教程的同时，您应该自己一点一点的添加代码来体会代码的作用，只有这样才可以真正理解这些概念。

从这里开始

在这篇教程里，我们将使用已经写好的代码作为模版。除了我们将要在 createScene 函数里面添加的代码之外，您可以暂时忽略其他的东西。在后面的教程里我会深入讲解 OGRE 程序是如何工作的，现在我们只需要从最简单的地方学起就行了。在您的编译器里创建一个新的工程，然后把下面的代码添加进去：

```
#include "ExampleApplication.h"

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

    ~TutorialApplication()
    {
    }
protected:
    void createScene(void)
    {
    }
};

#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"

INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine,
INT )
#else
int main(int argc, char **argv)
#endif
{
    // Create application object
    TutorialApplication app;

    try {
        app.go();
    } catch( Exception& e ) {
#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
        MessageBox( NULL, e.getFullDescription().c_str(), "An
exception has occured!", MB_OK | MB_ICONERROR | MB_TASKMODAL);
#else
        fprintf(stderr, "An exception has occured: %s\n",
                e.getFullDescription().c_str());
#endif
    }
```

```
    return 0;
}
```

假如您能够成功编译这段代码，您在运行的时候可以用 WASD 键移动，鼠标来转镜头，ESC 键来退出程序。

OGRE 是怎样工作的

这是一个很广的题目，我们将从场景管理器开始然后进一步了解场景节点和实体。这三个类是所有 OGRE 程序的基石。

场景管理器基础

在屏幕上显示的所有东西都是由场景管理器来管理。当您在场景中添加物体时，场景管理器会记录这些物体的位置。当您添加摄像机来观看某个场景时，场景管理器会记录摄像机的位置。当您添加平面、广告牌、灯光时，场景管理器同样会管理他们。 OGRE 里有很多种场景管理器。有的场景管理器渲染地面，有的场景管理器渲染 BSP 表等等。在后面，我们将进一步了解场景管理器。

实体基础

一个实体是可以在场景中渲染的物体之一。您可以把实体理解为任何一个 3D 模型。一个机器人可以是一个实体，一条鱼可以是一个实体，大地草原可以是一个非常大的实体。灯光，摄像机，粒子，广告牌等不能成为实体。
但在 Ogre 中你不能够直接将一个实体放入到场景中，而是将实体与场景节点绑在一起，这个场景节点则包括了实体的方位信息。

场景节点基础

场景节点将持续跟踪与它绑在一起的实体的方位。当你创建了一个实体时，它直到与一个场景节点绑定后才会被渲染。同样，一个场景节点也不能单独的在屏幕上显示出来，只有与一个实体绑定后才能在屏幕上显示。
场景节点可以绑定多个实体。例如在屏幕上有在行走的一个人物对象，并且希望这个对象产生发光效果，要实现这些，首先你需要创建一个场景节点，然后再创建一个人物对象的实体并与场景节点绑定在一起，之后你还需要创建一个光照模型也与这个场景节点绑定在一起。场景节点同样可以与其它场景节点绑定以描述更完整的对象。我们在后续的章节中介绍场景节点更多的用法。
在场景中，场景节点的位置总是与它的父节点相关。每一个场景管理器都包含一个根节点。

第一个 OGRE 程序

返回到我们刚才创建的代码，找到 TutorialApplication::createScene 成员函数。首先需要为整个场景设置环境光，这样才可以看到要显示的内容，通过调用 setAmbientLight 函数并指定环境光的颜色就可以做到这些。指定的颜色由红、绿、蓝三种颜色组成，且每种色数值范围在 0 到 1 之间。将下一句添加到 createScene 中：

```
mSceneMgr->setAmbientLight( ColourValue( 1, 1, 1 ) );
```

下一步创建一个 Entity ，通过调用 SceneManager 的 createEntity 方法来创建：

```
Entity *ent1 = mSceneMgr->createEntity( "Robot", "robot.mesh" );
```

变量 mSceneMgr 是当前场景管理器的一个对象，createEntity 方法的第一个参数是为所创建的实体指定一个唯一的标识，第二个参数 "robot.mesh" 指明要使用的网格实体，"robot.mesh" 网格实体在 ExampleApplication 类中被装载。
这样，就已经创建了一个实体，但还需要创建一个场景节点来与它绑定在一起。既然每个场景管理器都有一个根节点，那我们就在根节点下创建一个场景节点。

```
SceneNode *node1 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode( "RobotNode" );
```

这句代码首先调用场景管理器的 getRootSceneNode 方法来获取根节点，再使用根节点的 createChildSceneNode 方法创建一个名为 "RobotNode" 的场景节点。与实体一样，场景节点的名字也是唯一的。

最后，将实体与场景节点绑定在一起，这样机器人(Robot)就会在指定的位置被渲染：

```
node1->attachObject( ent1 );
```

编译并运行你的程序，在屏幕上你将看到一个机器人。

坐标和向量

在我们继续之前，需要先了解一下 OGRE 的坐标和向量。与其它图形引擎一样，OGRE 也使用 XZ 面作为其水平面，Y 轴作为纵轴。当你面对着屏幕时，从左至右的方向为 X 轴正方向，从下至上为 Y 轴正方向，从里至外为 Z 轴的正方向。

你现在可能注意到机器人正面朝 X 轴正方向，这也许是模型(mesh)设计时就预定好的方向。OGRE 并不会假定你的模型的方位，你所载入的每一个模型都可能有不同的“标准”方向。

OGRE 使用 Vectors 类来描述方向和位置，定义了 2 维、3 维及 4 维向量，其中 3 维向量使用得最多。如果你对向量的知识不熟悉，那建议你在正式使用 OGRE 之前先学习一些向量的知识，因为在复杂的程序设计中向量是非常重要的。

添加其它的对象

现在我们已经了解了 OGRE 中的方位了，让我们再次回到代码部分你就会发现在代码中并没有指定机器人的方向。实际上在 OGRE 中大量的函数都有自己默认的参数值，例如 SceneNode::createChildSceneNode 成员函数有三个参数：场景节点的名称、位置及方向，该成员的位置默认值为(0,0,0)。让我们创建另一个场景节点，这次我们指定它的方位信息:

```
        Entity *ent2 = mSceneMgr->createEntity( "Robot2",
"robot.mesh" );
        SceneNode *node2 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("RobotNode2", Vector3( 50, 0, 0 ) );
        node2->attachObject( ent2 );
```

这些代码看起来应该很熟悉，除了两个地方不同之外，其它部分与以前的代码都一样。首先我们创建的实体及场景节点的名称有细微的不同，其次创建场景节点时指定了节点的方位信息，我们让实体向 X 的正方向偏移了 50 个单位（这是相对与场景中的根节点而言，而所有的节点的位置都是相对于其父节点而言的）。编译并运行，你会发现屏幕上现在有两个并排着的机器人。

实体深入了解

实体类包含的内容相当的广，这里我不会向你介绍实体类的所有内容，只不过使你能够开始使用 OGRE 而已。不过我会向你介绍几个实体类有用的成员函数。

第一个是 Entity::setVisible 和 Entity::isVisible。利用这个函数，你可以设置任何一个实体为可见或不可见。当你想暂时隐藏一个实体时，与其销毁这个实体然后再在用的时候重新创建，不如简单的调用一下这个函数。注意你不需要“省着”用这些实体，任何一个对象的材质和贴图只会被载入到内存里一次，所以当你省系统资源的时候，你并没有真正节省多少。你唯一省掉的只不过是实体创建和毁灭的时间。

getName 函数返回实体的名称，getParentSceneNode 函数返回这个实体绑定的节点。

场景节点深入了解

场景节点类同样相当复杂。场景节点能实现很多事情，所以我们只了解一些常用的。

你可以利用 getPosition 或 setPosition 得到或者设定场景节点的位置 (总是和父节点相对的)。你可以利用 translate 函数来移动对象。

场景节点不仅仅决定一个对象的位置，它还可以控制一个对象的缩放比例和旋转角度。你可以用 scale 函数来设置一个对象的缩放比例，还可以用 yaw, roll, pitch 函数来旋转对象。你可以用 resetOrientation 来还原你对对象进行的所有旋转。你还可以用 setOrientation, getOrientation 和 rotate 函数对对象进行高级旋转。四元数会在后面的教程里对大家讲解。

你现在已经见过 attachObject 函数了。当你想对绑定到场景节点上的对象进行操作时，这些函数会很有帮助：numAttachedObjects, getAttachedObject(这个函数有很多版本), detachObject(同样很多版本), detachAllObjects。还有一大堆函数是来处理父节点和子节点的。

由于所有的移动都是相对于父节点的，我们可以很容易的使两个节点一起移动。我们现在已经有这些代码在我们的程序里:

```
	Entity *ent1 = mSceneMgr->createEntity( "Robot",
"robot.mesh" );
	SceneNode *node1 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode( "RobotNode" );
	node1->attachObject( ent1 );

	Entity *ent2 = mSceneMgr->createEntity( "Robot2",
"robot.mesh" );
	SceneNode *node2 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode( "RobotNode2", Vector3( 50, 0, 0 ) );
	node2->attachObject( ent2 );
```

假如我们将第 5 行从：

```
	SceneNode *node2 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode( "RobotNode2", Vector3( 50, 0, 0 ) );
```

改成：

```
	SceneNode *node2 = node1->createChildSceneNode( "RobotNode2",
Vector3( 50, 0, 0 ) );
```

这样一来 RobotNode2 就变成了 RobotNode 的子节点。移动 node1 会使 node2 跟着移动，移动 node2 却不会影响 node1。下面的代码就只会移动 RobotNode2：

```
node2->translate( Vector3( 10, 0, 10 ) );
```

下面的代码会移动 RobotNode，既然 RobotNode2 是 RobotNode 的子节点，RobotNode2 会跟着移动：

```
node1->translate( Vector3( 25, 0, 0 ) );
```

最后说一下，你可以使用场景管理器的成员函数 getSceneNode 和 getEntity 来获取场景节点和实体，这样你就不用保留你创建他们时使用的指针了。但你仍然应该保留很常用的实体的指针。

额外尝试

到目前为止你应该已经掌握了实体，场景节点和场景管理器的基础了。现在你可以试验一下下面的代码段：

缩放

你可以用 scale 这个成员函数来缩放网格。试一下变换 scale 里面的值看看什么效果：

```
        Entity *ent = mSceneMgr->createEntity( "Robot",
"robot.mesh" );
       SceneNode *node = mSceneMgr->getRootSceneNode()-
>createChildSceneNode( "RobotNode" );
       node->attachObject( ent );

       node->scale( .5, 1, 2 );

       ent = mSceneMgr->createEntity( "Robot2", "robot.mesh" );
       node = mSceneMgr->getRootSceneNode()-
>createChildSceneNode( "RobotNode2", Vector3( 50, 0, 0 ) );
       node->attachObject( ent );

       node->scale( 1, 2, 1 );
```

旋转

你可以用 yaw, pitch, roll 来旋转对象。Yaw 是 Y 轴的旋转，Pitch 是 X 轴，Roll 是 Z 轴。试一下变换角度(Degree)和合并多种旋转方式：

```
        Entity *ent = mSceneMgr->createEntity( "Robot",
"robot.mesh" );
       SceneNode *node = mSceneMgr->getRootSceneNode()-
>createChildSceneNode( "RobotNode", Vector3( -100, 0, 0 )  );
       node->attachObject( ent );

       node->yaw( Degree( -90 ) );

       ent = mSceneMgr->createEntity( "Robot2", "robot.mesh" );
       node = mSceneMgr->getRootSceneNode()-
>createChildSceneNode( "RobotNode2");
       node->attachObject( ent );

       node->pitch( Degree( -90 ) );

       ent = mSceneMgr->createEntity( "Robot3", "robot.mesh" );
       node = mSceneMgr->getRootSceneNode()-
>createChildSceneNode( "RobotNode3", Vector3( 100, 0, 0 ) );
       node->attachObject( ent );

       node->roll( Degree( -90 ) );
```

小结

恭喜你，现在已经掌握了 OGRE 的基石——实体，场景节点，场景管理器。你并不需要对上面所讲的那些函数都了如指掌。由于他们是很基础的东西，我们会经常使用。随着对 OGRE 了解的深入，你应该可以自然而然的掌握他们。

Ogre 环境配置

本课(甚至整个教程)涉及到的大部分文件(dll 和 cfg)都可以在 ${OgreSDK}/bin/debug 或 release 目录里找到。你创建的 debug 程序应该使用 debug 目录里的文件，而 release 程序应当使用 release 目录里的文件。

注意，在这里多数情况是指在 Windows 环境下的。在 Linux 中，相同的内容基本上是适用的，但共享库是以.so 作后缀，且存在于不同的位置，部分内容可能存在细微的差异。如果你遇到任何问题，请在 OGRE help 论坛求助。

动态链接库(DLLs) 与 插件(Plugins)

好，我们已经稍微玩转了一下 Ogre 的开发环境，我想进一步说明在一般情况下 OGRE 库是如何工作的，以及告诉您如何更轻松的使用它。

Ogre 被分为若干类共享库。

第一类是原始库，以及它的依赖库。Ogre 库全都包含在 OgreMain.dll 里。这个 dll 还需要其它的一些库，比如 cg.dll。这些 dll 必须毫无例外地包含在每一个 ogre 应用中。

第二类库是插件。Ogre 将很大一部分功能抽到这些库里面，以方便能根据具体需要来开启或关闭这些功效。Ogre 包含的基本插件拥有以"Plugin_"开头的文件名。如果需要，你能亲自写一个插件，但我们的教程将不涉及这些内容。Ogre 还为渲染系统提供插件(如 OpenGL、DirectX 等)。这些插件以"RenderSystem_"为前缀。有了这些插件，你就可以方便地在应用程序中添加或者移除渲染系统。这是非常有用的，如果你正在写一个 OpenGL 专门的着色程序，并希望当运行在 DirectX 上时不执行它，你则只要移除这个渲染系统，它就不存在了。另外，如果你致力于一个非标准的平台，你可以写你自己的渲染系统插件，但我们的教程不涉及。我们将在下一课告诉你如何移除插件。

第三类库是第三方库和帮助库。Ogre 本身只是一个图形渲染引擎，它并不包含像 GUI 系统、输入控制、物理引擎这类东西，你需要其它第三方帮助库。CEGUI 库就是一个能轻松整合进 Ogre 的 GUI 系统，以"CEGUI*"开头的 dll 文件和"OgreGUIRenderer.dll"就是它的一部分。我们将在以后的教程里介绍 CEGUI。鼠标键盘输入则通过 OIS 来实现(一个输入系统)，它包含在 OIS.dll 里。还有其它很多能够提供更多功能的库（不包含在 SDK)，比如音效和物理。你能在 Wiki 或论坛里找到更多的信息。

总之准则就是，当你在本地调试你的程序时，你可以将所有的功能都“打开”（也就是说，不移除任何东西）。当你准备发布你的程序时，你应用 Release 模式构建它，包括所有你用到了的 dll 文件。而且你应该移动那些你没有使用的 dll。如果你的程序没有用到 CEGUI，而是使用了 OIS，就不要包含 CEGUI 进来，但你必须包含 OIS 的 dll，否则的你程序跑不起来。

配置文件

Ogre 的运行需要几个配置文件。它们用来控制哪些插件需要加载，程序的资源在哪里定位等等。我们来简单地看一下每个配置文件是用来做什么的。若你有更特殊的问题，请直接来 Ogre help 论坛询问。

plugins.cfg 这个文件指定应用程序使用的插件。如果你要在程序中添加或移除某个插件，就要修改这个文件。要去掉某个插件，只需要删除某一行，或用#直接注释掉它。你想要添加一个插件，就要添加像"Plugin=[PluginName]"这样的一行。注意，你不要在插件名后面加上.dll 后缀。你的插件同样不必以 "RenderSystem_" 或 "Plugin_" 开头。 通过修改"PluginFolder"变量，你还可以定义 Ogre 搜索插件的目录。你可以使用绝对或相对路径，但不能使用像 $(SomeVariable)这样的环境变量。

resources.cfg 这个文件含有 Ogre 搜索资源的目录列表。资源包括：脚本、模型、纹理等等。你同样可以使用绝对或相对路径，但你不能使用如 $(SomeVariable)环境变量。注意，Ogre 不会搜索子目录，所以你如果有多层目录必须手工输入它们。比如，你有一个目录树，像"res\meshes" and "res\meshes\small"，你就要为这些路径添加两个入口。

media.cfg 这个文件告诉 Ogre 更多关于某些资源的细节。目前你不太可能改动这个文件，所以我们跳过去。你能在手册或 Ogre 论坛里找到更多的信息。

ogre.cfg 这个文件是 Ogre 的配置窗口生成的。这个文件与你的个人电脑和显示配置相关。当你把应用程序发布给别人时，不必配提供这个文件，因为他们的设置很可能不同。注意，你不能直接编辑这个文件，而要使用那个配置窗口。

quake3settings.cfg 这个文件是与 BSPSceneManager 一起使用的。除非你使用这个场景管理器(目前你不会使用)，否则你不需要这个文件，我们忽略它。同样地，除非你正在使用 BSPSceneManager，你不要把这个文件与你的程序一起发布，就算那样，它也可能会完全不一样，这要看你程序的需求。

以上是所有 Ogre 需要操作的文件。Ogre 要能找到"plugins.cfg", "resources.cfg", 和 "media.cfg"，它才能正确运行。在后续的教程里，我们将会深入介绍这些文件，以及如何改变它们的地址，用它们来做更高级的事情。

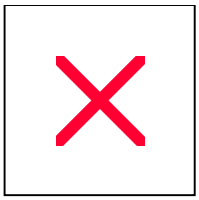
一个更好的调试程序的方法

注意，这个部分是 Windows 和 Visual C++相关的。

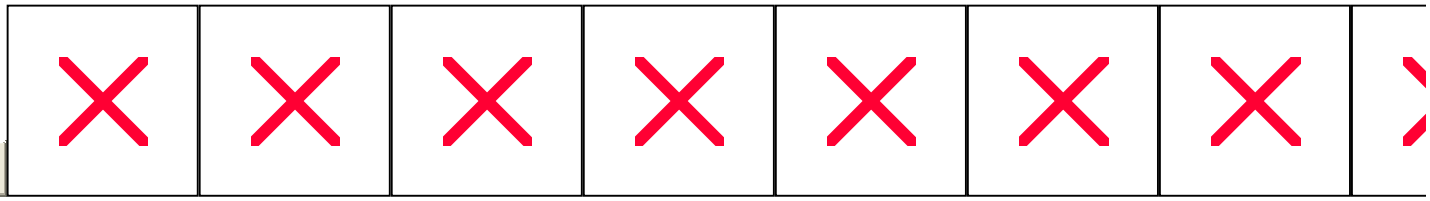
如很多地方讲述的，Ogre 必须要能找到配置文件、dll 文件以及程序所用到的 media 资源文件。为了解决这个问题，大多数人手工地把 bin 目录从 OgreSDK 拷到他们的工程目录，来保证正确的 dll 文件与可执行文件在相同的目录。如果你正在创建一个游戏并准备发布给别人，这可能是最好的办法，因为你毫无疑问地会修改这些配置文件，从标准内容中添加删除插件。而在其它一些情况里，持贝所有的 dll 文件到每一个 Ogre 工程里浪费了空间和时间。其实你可以有另一种办法。

另一种方法就是把 OgreSDK 里的 dll 文件拷贝到 c:\Windows\System 目录。这样做的优点就是，不管理你的可执行文件在哪，Windows 总能找到合适的 dll。要让这样做生效，你还要相应地修改 resources.cfg 和 plugins.cfg 来包含 media 目录和插件的绝对路径。现在，无论何时你创建了一个工程，你只要把修改过的配置文件拷贝到你的 bin\debug 和 bin\release 目录就行了。我个人不太使用这种办法，因为这有可能让人分不清在 windows 目录里哪些是 OgreSDK 的 dll 文件。Ogre 新版本出得很快，所以实际上手工地去用这种方法去安装是非常烦琐的。

一个更好的选择就是，让那些 OgreSDK 文件原地不动，而把每个 Ogre 应用的工作目录(working directory)设置成 OgreSDK 的 bin\release 或 bin\debug 目录。要这样做的话，找到你的工程属性，把工作目录(working directory)改成 "C:\OgreSDK\bin\$(ConfigurationName)"。你要把 "C:\OgreSDK" 改成你自己的 Ogre 安装目录。好了，不需要拷贝任何文件，你的 Ogre 程序就能运行了。反正我是用这个方法的。唯一的缺点就是，若你修改了配置文件，你所有的 Ogre 工程都会受到影响，这显然是糟糕的。如果你使用了这个方法，而又想为你的工程修改配置文件的话，你最好还是照以前那样把文件复制到工程里，然后把工作目录给改回来。

our search terms 

search form

earch 

工具箱

- 链入页面
- 链出更改
- 特殊页面
- 可打印版
- 永久链接

Google Adsense